THE UNIVERSITY OF SUSSEX

Environmentally Robust Multiple Camera Tracking

Submitted for the degree of Master of Philosophy

October 2019

Samuel J Hartlebury

Declaration

I hereby declare that this thesis has not been submitted, either in the same or different form, to this or any other university for a degree.

Signed … … … … … … … … … … … …

Samuel J. Hartlebury

THE UNIVERSITY OF SUSSEX

The degree of Master of Philosophy

I Samuel J. Hartlebury author of this thesis Environmentally Robust Multiple Camera Tracking give permission to the Librarian of the University of Sussex to make available for inter-library loan and for photocopying, the above mentioned thesis which is deposited in the University Library. I agree that the abstract of the thesis may be published in such lists of sources as may be approved from time to time by the Librarian.

Signed … … … … … … … … … … …

Date … … … … … … … … … … …

Acknowledgements

I would like to firstly thank my supervisors Professor Chris R. Chatwin and Dr. Rupert C.D. Young for their help and supervision. I am very grateful to Dr. Phil M. Birch for his help and support throughout this research and steering me in the right directions. I would also like to acknowledge the support and funding from the Engineering and Physical Sciences Research Council, without it this research would never have come to be.

Finally, I would like to say thank you to my parents for the continuous love, support and encouragement of my studies.

THE UNIVERSITY OF SUSSEX

Environmentally Robust Multiple Camera Tracking

Submitted for the degree of Master of Philosophy

October 2019

Samuel J Hartlebury

Abstract

A significant growth of the use of surveillance cameras has arisen from both the availability of low-cost home security and post-September 11th security measures. With such a plethora of surveillance cameras available and already in use, tracking a person or object from one field of view to another accurately is a challenging possibility; recognising the same person at different spatial locations, under different lighting conditions, at different scales and orientations. In order to address these challenges and provide a solution, a review of recent and past literature is provided.

The main theme of this research is investigating methods to improve tracking of objects and people in dynamic environments and applying computational techniques to provide solutions to optimise such tracking systems. Image processing

techniques are explored and refactored to adapt to currently available single-board computing power. Optimisation methods for speed of computing are investigated, presenting the paradigm of parallel programming during the design of "computationally intense" algorithms. The research also addresses cross-platform software/ server application design.

In controlled environments current tracking systems perform well, however, this project explores methods to take multiple camera tracking to a higher level where they can, in real time, robustly cope with: rapid changes in lighting and track objects between indoor and outdoor scenarios at any time of day or in any weather conditions, severe image occlusion, rapid changes in direction, orientation and velocity of the object being tracked and be invariant to image clutter and noise. Thus the outputs are twofold: track a human/object across multiple cameras and ensure the algorithm is fast enough to run in real time on a modern processor.

This research explores algorithms to deliver colour illumination invariance, also known as colour constancy. Colour illumination invariance can be applied as a pre-processing step to all cameras in a multi-camera environment. The research also investigates experimental assessment of multi-camera performance, focusing mainly on robustness to environmental changes.

There are three main objectives for a tracking algorithm being used in the proposed system. Firstly, the tracking algorithm must accurately detect objects independently of their scale change and rotation. Secondly, the tracking algorithm must accurately detect objects across multiple cameras in different lighting conditions. The third objective for the tracking algorithm is that it must be able to attain a high level of colour constancy. The last objective can be implemented as a

pre-processing step to such a tracking algorithm. This research explores the use of the Scale Invariant Feature Transform (SIFT) and the Speeded-Up Robust Features (SURF) algorithm. These algorithms are discussed in detail in the literature review as well as methods for providing colour illumination invariance.

Table of Contents:

List of Acronyms

2

2D
two dimensional / dimension · 52

3

3D
Three dimensional / dimensions · 4

A

ABC
Abstract class · 24
ADT
Abstract data types · 22

B

BRIEF
Binary Robust Independent Elementary Features · 87
BRISK
Binary robust invariant scalable keypoints · 87

C

C++
a high-level programming language developed by Bjarne Stroustrup at Bell Labs · 18
CIE L*A*B*
also known as CIE L*a*b* or sometimes abbreviated as simply "Lab" color space) is a color space defined by the International Commission on Illumination (CIE) in 1976. It expresses color as three values: L* for the lightness from black (0) to white (100), a* from green (−) to red (+), and b* from blue (−) to yellow (+) · 29
CPU
Central processing unit · 4

D

d.p.
decimal places · 44
D65 illuminant
Standard illuminant defined by the International Commission on Illumination · 33
DBGW
Database Grey World · 8
DSLR
Digital single-lens reflex · 33
DVD
Digital versatile disc · 22

E

F

G

L

M

O

P

proc
: procedure · 68

P-SURF
: Parallel Speeded-Up Robust Features · 4

Python
: an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects · 53

Q

Qt
: a cross-platform application development framework for desktop, embedded and mobile · 18

R

rgb
: Normalised RGB · 15

RGB
: Red, green and blue · 8

Ruby
: an interpreted, high-level, general-purpose programming language · 53

S

SIFT
: Scale Invariant Feature Transfrom · 2

SLAM
: Simultaneous localisation and mapping · 4

SMC
: Sequential Monte Carlo · 6

SoG
: Shades of Grey · 44

SURF
: Speeded-Up Robust Features · 2

T

TV
: Television · 22

V

VCR
: Video cassette recorder · 22

List of Figures

List of Tables

List of Equations

Chapter 1 : INTRODUCTION

1.1 Object recognition and robust tracking

Object recognition is a computer vision technique for identifying objects in images or videos. Object recognition is a key output of artificial intelligence and machine learning algorithms. When humans look at a photograph or watch a video, we can readily spot people, objects, scenes, and visual details. The goal is to teach a computer to do what comes naturally to humans: to gain a level of understanding of what an image contains. Object recognition is a key technology behind driverless cars, enabling them to recognize a stop sign or to distinguish a pedestrian from a lamppost [1] . It is also useful in a variety of applications such as disease identification in bio-imaging, industrial inspection, and robotic vision [2] [3] [4].

Object detection and object recognition are similar techniques for identifying objects, but they vary in their execution. Object detection is the process of finding instances of objects in images [5]. In the case of machine learning, object detection is a subset of object recognition, where the object is not only identified but also located in an image. This allows for multiple objects to be identified and located within the same image [6].

1.2 Research Outputs

In controlled environments current tracking systems perform well, however, this thesis outputs methods to achieve multiple camera tracking in real time and robustly cope with: rapid changes in lighting, track objects between indoor and outdoor scenarios at any time of day or in any weather conditions, severe image occlusion, rapid changes in direction, orientation and velocity of the object being tracked and be invariant to image clutter and noise.

The initial proposed area for research is exploration of algorithms to deliver colour illumination invariance, also known as colour constancy. Colour illumination invariance can be applied as a pre-processing step to all cameras in a multi-camera environment. The second area for research is experimental assessment of multi-camera performance, focusing on mainly on robustness to environmental changes.

There are three main objectives for a tracking algorithm being used in the proposed system. Firstly, the tracking algorithm must accurately detect objects at a variation of scales and rotation. The next objective is that the tracking algorithm must accurately detect objects across multiple cameras in different lighting conditions. The final objective for the tracking algorithm is that it must be able to attain a high level of colour constancy. The last objective can be implemented as a pre-processing step to such a tracking algorithm.

In recent literature there are two main object recognition and tracking algorithms that already meet the above mentioned objectives, these are the Scale Invariant Feature Transform (SIFT) and the Speeded-Up Robust Features (SURF) algorithm. These algorithms are discussed in detail in the literature review as well as methods for providing colour illumination invariance.

Chapter 2 : Literature Review

2.1 Methods for object recognition and tracking

2.1.1 Scale Invariant Feature Transform (SIFT)

There are many different methods for recognition and tracking of objects both in image and video processing, the most well-known being David Lowes Scale Invariant Feature Transform (SIFT) [7]. The majority of other algorithms use similar approaches to SIFT, detecting distinctive features invariant to scale or rotation, but aim at improving computational speed for real-time applications [8]–[12].

SIFT is a method of extracting information from an image to describe object properties such as scale, rotation and skew. The SIFT feature extraction process is arranged into 4 stages; Scale-space extrema detection, Keypoint localisation, Orientation assignment and Keypoint descriptor extraction [7]. Scale-space extrema detection is achieved using a difference of Gaussian function to detect intensity contrast for keypoint selection across an image at all locations, invariant to rotation and scale. The next stage is localisation of the keypoint descriptors. Keypoints for

extraction are tested to determine scale and orientation and then selected depending on their stability. One or more orientations are then assigned to each keypoint location based on local image gradient directions. All operations thereafter are performed on image data that has been transformed relative to the assigned orientation, scale and location for each feature, thereby providing invariance to these transformations [7]. The image intensity gradients are measured at different scales in the region around each keypoint. These are transformed into a representation that allows for levels of object shape distortion [7].

SIFT keypoints are useful due to their distinctiveness making it possible to find correct matches with other keypoints stored from previous image transformations. This distinctiveness is attained by creating a multi-dimensional vector that describes the illumination gradients direction and scale within an area of the image [7]. The keypoints have been shown to be invariant to image rotation and scale and are robust across a large range of affine distortion, noise and change in illumination [7], [13]. Large numbers of keypoints can be extracted from a set of images leading to robustness in finding small objects among clutter.

Keypoints are detected over a range of scales meaning that small local features are able to be used for matching small and highly occluded objects while large keypoints can be used for images containing noise or blur. Several thousand keypoints can be extracted from an image with near real-time performance on standard PC hardware [7]. The keypoint matching method described by David Lowe uses nearest-neighbour search, a Hough transform for identifying clusters that decide on object positioning, least-squares positioning determination and final verification [14]–[16].

Other applications currently using SIFT keypoints include view matching for 3D reconstruction, motion tracking and segmentation, robotics simultaneous localisation and mapping (SLAM) and panorama image stitching [7], [17], [18]. There are many parts of the SIFT process that could be explored further for deriving more capable invariant and distinctive image features, for instance the features described use only the contrast in image intensity to derive keypoints , so further distinctiveness could be derived by including illumination invariant colour descriptors [7], [19]–[21].

In the original paper describing SIFT by David Lowe, the descriptors are also claimed to be invariant to change in illumination [7]. This is true of slight changes in illumination, however, during changes in illuminant colour, contrast gradients can seemingly disappear altogether due to changes in surface reflectance [22]–[25]. If such gradients were to be lost, SIFT would fail to make correct matches. Joost van de Weijer et al explain how colour descriptors can be obtained from images on a per channel basis, instead of just using intensity gradients [20], [26].

2.1.2 Speeded-Up Robust Features (SURF)

As mentioned previously, there are many other feature extraction techniques similar to SIFT with the aim of speeding up the algorithm. The Speeded-Up Robust Features algorithm (SURF) can be computed much faster than SIFT by relying on integral images for image convolutions instead of building a difference of Gaussians pyramid for detecting scale-space extrema [8]. SURF uses a Hessian matrix-based measure for detection of features and a distribution based descriptor which has been shown to

be more than three times faster than using a difference of Gaussian and five times faster than Hessian-Laplace methods [8], [27].

Parallel Speeded-Up Robust Features (P-SURF) is a CPU-based parallel method for computing the Speeded-Up Robust Features (SURF) algorithm. In the original serial computation of SURF, there are seven major steps to extract and represent interest points from an image [8], [27], [28]. The parallelised method decomposes the processes in each of the seven steps and assigns each thread a part of the process so that they can be computed on separate processors to gain an overall speedup in computational time spent. This method of feature tracking can be used to satisfy the second output of this research; to ensure the algorithm is fast enough to run in real time on a modern processor.

The last four steps in the serial algorithm are merged in order to reduce the overhead of managing threads should they be separated. The scale-space analysis is not merged with the subsequent processing because it has to be completed before the localisation can start. In the computation of P-SURF the number of coexisting threads should not exceed the number of available processors in the system as this could have the opposite effect and slow down the process instead of speeding it up [28].

Each thread, including the main thread, is initially assigned a part of the process. Once a thread is finished it will return its output to the main thread and terminate itself. A new thread will be assigned a new part of the process and so on until there are no more processes to assign. The main thread is bound to its starting processor and each new thread is bound to an available processor that is not active with any threads performing a part of the task.

Experiments showed that this fixed allocation of processors produced better results in terms of both performance and stability than if the threads are left to the scheduling of the kernel for processor assignment [28]. The performance was comparable to some GPU-based implementations and can operate on platforms more commonly available (i.e. windows). On a system equipped with an Intel Core Duo P8600 at 2.4GHz, P-SURF was able to run at 33 frames per second on a randomly selected 640×480 image [28].

The image loading method is parallelised by creating multiple copies of the same file input stream, while the APIs for image loading found in most image manipulation libraries are sequential [28]. The original creators of SURF in their paper state that further speedup could be gained if the scale-space analysis is performed in parallel [9] [28].

2.1.3 Particle Filters

Particle filters or Sequential Monte Carlo (SMC) methods are a set of genetic-type particle Monte Carlo methodologies to solve a filtering problem. The term "particle filters" was first introduced in 1997 by Del Moral [29]. Particle filters are used for searching by generating a set of randomly distributed particles attached with heuristic data describing each particles position in a multidimensional search space. A filter is then applied to each particle in the search space, retrieving strong matches between the particles and the target heuristics. Similar to genetic programming, the particles with the most error are removed. The particles that have very high

probabilities are used for resampling, adding another generation of particles in the highly probable locations of the search space for refinement.

Particle filters have been used widely in computer vision for tracking specific target features in an image. A simple approach to tracking with particle filters can be achieved using colour histogram analysis and has been shown to be highly robust in cases of fast moving targets and scene clutter [30]. However, yet again we face the colour constancy problem; targets being tracked using colour histograms would have a different “signature” under different lighting conditions and shadows [31]–[33].

Recent literature shows how using SIFT and SURF features in a particle filter instead of colour histograms can greatly improve tracking under changing lighting conditions [32], [34]–[36]. This is mainly due to the robustness of SIFT and SURF features under changes in scale, rotation and luminance [7], [27], [35]. In the previous chapter we see how using a colour constancy algorithm as a pre-processing step can greatly enhance the accuracy of SURF feature matching. Applying this knowledge to a particle filter tracking system using SIFT or SURF features would be seen as being greatly beneficial.

2.2 The Colour Constancy Problem

A fundamental problem in digital image acquisition is colour constancy. An object may appear to be of a certain colour depending on three factors; the properties of the camera being used, the physical properties of the surfaces within a scene and the angle from which the light source is reflected from a surface (incident angle)

[37][38][39][40]. Overcoming this problem would be highly beneficial to object recognition and feature detection algorithms due to the fact that an object that appears to be of a certain colour in one instance, often due to lighting inconsistencies, could appear to be of a completely different colour in another scene.

A simplified model of image formation is described below:

$$P_k = \int S(\lambda)E^o(\lambda)Q_k(\lambda)d\lambda \quad [37][24][38][25]$$

Equation 2.1: A simplified model of image formation

where $S(\lambda)$ defines the surface characteristics at a particular spatial location: it defines the proportion of light incident at that position that is reflected on a per wavelength (λ) basis. $E^o(\lambda)$ is the spectral power distribution of the scene illuminant; it characterises how much energy the source emits as a function of wavelength. $Q_k(\lambda)$ is the spectral sensitivity function of the imaging device sensor, which determines what proportion of light energy incident upon it is absorbed at each wavelength. Thus the sensor response P_k is a measure of the total energy absorbed by the sensor over the range of wavelengths to which the sensor is sensitive. The subscript k distinguishes a particular class of imaging sensor [37][38].

2.2.1 The Grey-World Hypothesis

There are many colour constancy algorithms available, each approaching the colour constancy problem with a different strategy. The most widely used algorithm,

defined by Finlayson [37] as being the standard colour constancy method, is the Grey-World algorithm.

The Grey-World algorithm assumes that the mean reflectance in a scene is achromatic and estimates the light source colour from the average RGB pixel value. Using this light source estimate, the Grey-World algorithm multiples each channel to adjust the scene to be achromatic (white light), applying a diagonal transform in the colour space. One method of improving the Grey-World algorithm was introduced by Gershon et al. coined Database Grey-World (DBGW)[22]. If the images under evaluation are part of a coherent image data base, Gershon et al. [22] showed that assuming the average of a scene to be equal to the average reflectance of the database, improves the results over the standard grey-world method. As an example, they mention forest pictures full of green colours. In this case, most colour constancy methods will predict light sources biased towards the green colour. The database-compensated grey-world algorithm resolves this problem. Gershon et al. [22] have also shown that the spatial average, computed in the estimation stage of the Grey-World algorithm, is biased towards large surfaces. They proposed a modified algorithm which removes this problem by segmenting the image into patches of uniform colour before estimating the illuminant. The sensor response from each segmented surface is then counted only once in the spatial average so that surfaces of different sizes are given equal weight in the illuminant estimation stage. Another limitation of the Grey-World algorithm is that it does not take into account the surface characteristics or the sensor properties of the camera [23].

2.2.2 Max-RGB

Another popular colour constancy method is called max-RGB . It is based on the assumption that the reflectance for each of the three channels is equal [41]. This method is sometimes explained as being derived from the white-patch hypothesis [38], [39]. Since a white patch reflects all the incident light, its position in the image can be found by searching for the maximum RGB values. It should be noted however that the max-RGB method does not require the maxima of the separate channels to be on the same location, hence it also obtains correct illuminant estimation results when the maximum reflectance is equal for the three channels [38], [39].

The Grey-World and Max-RGB algorithms are used widely in digital cameras as it uses a small amount of computational power [25], computing the average reflectance of a scene (Grey-World) or the maximum reflectance of a scene (Max-RGB).

Finlayson and Trezzi [42] showed that with minor adaptations results are obtained which are similar to those of complex colour constancy algorithms. In fact, they showed that the max-RGB method and the Grey-World method can be interpreted as the same algorithm applied with different instantiations of the error function. The max-RGB method is shown to be equal to applying the $L\infty$ Minkowski norm and Grey-World is equal to using the L1 norm. They further show that the best colour constancy results are obtained with the L6 norm. Although these simple colour constancy algorithms are slightly outperformed by more elaborate methods, e.g. gamut mapping (reviewed later), they perform surprisingly well while they are conceptually simpler [38].

2.2.3 The Grey-Edge Hypothesis

Joost van de Weijer et al propose a novel approach to colour constancy called Grey-Edge, based on estimating the scene illuminant from the derivative structure of images. This differs from the Shades of Grey, Grey-World and Max-RGB algorithms, whose illuminant estimates are computed from the zero-order structure of images [38]. This method takes pixels from surface edges in the image and uses that as a basis for estimating the light source, assuming the edge of a surface reflects the light source and assuming that the average edge difference in a scene to be achromatic [38].

2.2.4 Gamut Mapping

One of the most successful colour constancy methods is gamut mapping proposed by Forsyth [38], [41], [43], [44]. Gamut mapping is based on the observation that only a limited set of RGB values can be observed under a given illuminant [43]. The set of all possible RGB values for the canonical illuminant, typically a white illuminant, is called the canonical gamut [38], [43]. This canonical gamut is proven to be a convex hull in RGB space [43], [45]. The algorithm computes what transformations map an observed gamut into the canonical gamut. From these transformations, the illuminant colour is derived. The gamut mapping algorithm provides among the best results in

colour constancy experiments [44]. Finlayson et al. [37], [39] improve the gamut mapping algorithm by restricting the transformations to be plausible, meaning that only illuminants are allowed which correspond to existing illuminants. This adaptation of the gamut algorithm, called GCIE for gamut constrained illumination estimation, was shown to outperform the standard gamut algorithm

A similar colour constancy method has been proposed by Barnard [44] selecting the appropriate transformation from the feasible set of transformations computed with the Gamut mapping method. In the original work, Forsyth [43] proposed to take the transformation belonging to the gamut with the maximum volume. Instead, Barnard [44] considered various exponentials of the geometric mean of the transformation vector to select the best transformation from the feasible set. He showed that by varying the exponential, the selection criterion changes from taking the average over all transformations to the maximum volume heuristic. Like in the case of the Shades of Grey method, intermediate exponentials were shown to obtain better results [25].

Finlayson [39] recognised that features such as shape and shading affect the magnitude of the recovered light but not its colour. To avoid calculating the intensity of the illuminant Finlayson carried out computation in a 2D chromaticity space. If we characterise image colours and illuminants by their chromaticities we can define a matrix with elements set to one when chromaticity can be seen under a certain illuminant and set to zero if not [25] [45]. One problem with gamut mapping is that not all chromaticities correspond to possible illuminants. To overcome this problem the columns of the matrix can be restricted to those corresponding to possible light colours [46] [39] [45]. Compared to the Minkowski norm based colour constancy algorithms (Grey-world, Max-RGB, Shades of Gray), the gamut mapping method is

computationally expensive, thus not practical in real-time applications [38], [41], [44].

2.2.5 Colour Temperature Estimation

It has be shown by Yang and Sohn [47] that being able to estimate the colour temperature can dramatically increase the accuracy of colour constancy algorithms. The algorithm below describes the image acquisition model including the Lambertian shading constant which can be manipulated to estimate colour temperature.

Image Acquisition Model (Lambertian [48]):

$$P_x^i = \sigma \int E(\lambda)\, S^i(\lambda)\, Q_x(\lambda)\, \Delta\lambda \quad [47]\ [23]$$

Equation 2.2: Lambertian Image Acquisition Model

$x \in$ R, G, B
i = Pixel of image data
σ = Lambertian shading constant
E = Spectral power distribution of light
λ = Wavelength
S = Surface reflectance
Q = Spectral sensitivities of camera sensors

The spectral sensitivities of the camera sensors can be estimated and modelled using delta functions such that:

$$Qx(\lambda) = \delta(x - \lambda x)$$

Equation 2.3: Spectral sensitivities of camera sensors modelled using delta functions

λx = Sensitive wavelength of each sensor
$x \in q_R, q_G, q_B$ used as scaling factors

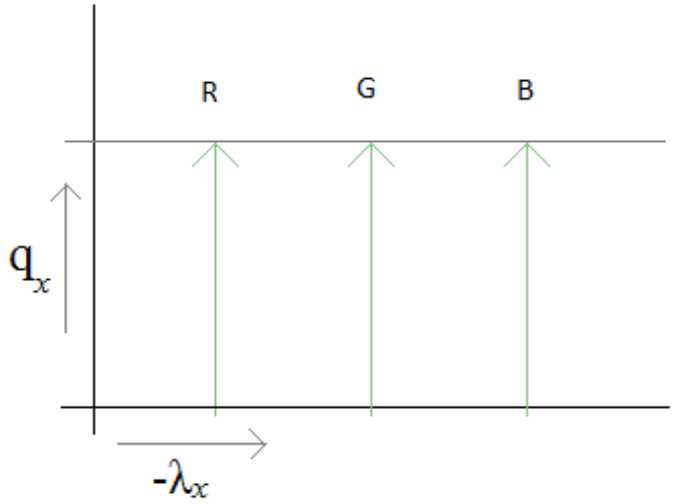


Figure 2.1: Delta functional of RGB frequency responses with q_x as scaling factor and λ_x as the sensitive wavelength of each sensor.

The Temperature of a light source can be calculated using Planks Law with the assumption that the spectral power distribution of light follows the spectral power distribution of a black body [49][47] (using the term 'A' as a scaling factor).

$$E_{(\lambda)} = A\frac{2hc^2}{\lambda^5}\frac{1}{\frac{hc}{e\lambda kT} - 1}$$

Equation 2.4: Planck's law for energy radiated per unit volume by a blackbody

[49][47]

This formula can then be simplified as follows:

$$c_1 = 2hc^2$$
$$c_2 = hc\ /\ k$$

$$E(\lambda) = A\frac{C_1}{\lambda^5}\frac{1}{\frac{C_2}{e\lambda T} - 1}$$

$$E(\lambda) \approx Ac_1\lambda^{-5}e^{-\frac{C_2}{\lambda T}}$$

Equation 2.5: Simplification of Planck's law by Son and Yang

[49][47]

The image acquisition process can there for be expressed as :

$$p_x^i = \sigma A c_1 \lambda_x^{-5} e^{-\frac{c_2}{\lambda_x T}} S^i(\lambda_x) q_x \ , \ x \in \{R, G, B\}$$

Equation 2.6: Image Acquisition process expressed from simplification of Plank's law

[49][47]

To simplify the formula further, as a pre-processing step the red and blue channels are divided by the green channel. To gain a more accurate estimate of the colour temperature it is best if the temperature is estimated both with the red and blue divided by the green channel and the red and green divided by the blue and average the result from both. The natural logarithm of the simplified expression is then taken to extract the term 'T' such that:

$$\ln\left(\frac{p_x^i}{p_G^i}\right) = \ln\left(\frac{\lambda_x^{-5} e^{-\frac{c_2}{\lambda_x T}} S^i(\lambda_x) q_x}{\lambda_G^{-5} e^{-\frac{c_2}{\lambda_G T}} S^i(\lambda_G) q_G}\right) = \ln\left(\frac{\lambda_x^{-5} S^i(\lambda_x) q_x}{\lambda_G^{-5} S^i(\lambda_G) q_G}\right) + \frac{C_2}{T}\left(\frac{1}{\lambda_G} - \frac{1}{\lambda_x}\right) \ , x \in \{R, B\}$$

Equation 2.7: Natural logarithm of the simplified expression to extract the temperature (term "T")

[47]

$S^i(\lambda_x)$ is a constant (for each channel of "x") determined by the camera characteristics and therefore remains the only unknown set of variables (for R,G,B). However, $S^i(\lambda_x)$ can be removed by taking an expectation operator (E[·] the average of all the pixel values in an image) and using the Grey-World methodology [47] [5] as follows:

$$E\left[\ln\left(\frac{p_x^i}{p_G^i}\right)\right] = E\left[\ln\left(\frac{\lambda_x^{-5}S^i(\lambda_x)q_x}{\lambda_G^{-5}S^i(\lambda_G)q_G}\right)\right] + E\left[\frac{C_2}{T}\left(\frac{1}{\lambda_G}-\frac{1}{\lambda_x}\right)\right]$$

$$\ln\left(\frac{E[p_x^i]}{E[p_G^i]}\right) = \left[\ln\left(\frac{\lambda_x^{-5}q_xE[S^i(\lambda_x)]}{\lambda_G^{-5}q_GE[S^i(\lambda_G)]}\right)\right] + \frac{C_2}{T}\left(\frac{1}{\lambda_G}-\frac{1}{\lambda_x}\right)$$

$$\because E\left[S^i(\lambda_R)\right] \approx E[S^i(\lambda_G)] \approx E[S^i(\lambda_B)]\ [47]$$

$$\ln\left(\frac{E[p_x^i]}{E[p_G^i]}\right) = \ln\left(\frac{\lambda_x^{-5}q_x}{\lambda_G^{-5}q_G}\right) + \frac{C_2}{T}\left(\frac{1}{\lambda_G}-\frac{1}{\lambda_x}\right), x \in \{R,B\}$$

Equation 2.8: Removal of camera sensitivity from simplified equation using expectation operator and grey-world assumption

$$\frac{1}{T} = \frac{1}{C_2}\left(\frac{\lambda_x\lambda_G}{\lambda_x-\lambda_G}\right) \times \ln\left(\frac{\lambda_G^{-5}q_GE[p_x^i]}{\lambda_x^{-5}q_xE[p_G^i]}\right), x \in \{R,B\}$$

Equation 2.9: Simplified expression with camera sensitivity removed, in terms of Temperature "T"

[47]

This method of estimating the colour temperature is shown by Yang and Sohn [47] to improve the illuminant estimate. The test images used in Fig.4 [47] clearly displays this as it shows colour temperature estimation vs. the Grey-World assumption on a set of test data:

[47]

Figure 2.2: Comparison of illumination compensated images.Top to bottom: test image, image rendered by our proposed method, image rendered by GWA-based conventional method.

In the last column of test data it is clear to see where the Grey-World algorithm fails, compared to the method described above, in estimating the correct illuminant as the white paper is seen to be incorrectly adjusted to become a tinge of blue.

2.2.6 Spectral Sharpening

Spectral sharpening is a method of transforming pixel values of multispectral images produced by a colour camera, scanner, or other optical device into new values that would have resulted from sensors with more narrowband spectral sensitivities [50]. The utility of such a transform is that for many computer vision and colour image processing algorithms, sharper sensors result in better performance. For example, consider the simplest form of colour correction, the von Kries [51] diagonal transform for correcting from red/green/blue (RGB) values under one illuminant to those under a second illuminant [50]. Theoretical sensors that act as delta functions would exactly obey a diagonal transform and it has been shown that spectral sharpening could greatly benefit such a colour constancy strategy. However, since spectral sharpening need not produce positive sensor curves, it is entirely possible that transformed RGB triples may include negative values. This presents no problem if a particular algorithm may use negative values in the transformed space, since one can then simply transform results back using an inverse transform [52] [50].

Unconstrained optimisation gives the best energy concentration but results in curves with negative lobes. An optimisation based on constraining only transformed RGB's rather than the curves themselves does best for a constrained sharpening for the camera studied, delivering almost as good results as for unconstrained sharpening but without the penalty of negative colours. The sharpened curves found are close to the best curves that result from unconstrained sharpening [52] [50].

If coefficient multipliers of sensor sets are constrained to be nonnegative, then the solution with the most concentration of energy in a sharpening interval is necessarily

one of the original sensors themselves. Optimisation with constrained sensors did better than optimisation with constrained coefficients for any of the L1- or L2-based schemes [52] [50].

2.3 Colour Spaces

2.3.1 Normalised Colour Spaces

Invariant values can be obtained by normalising RGB values by their intensity (Intensity = R+G+B) resulting in the *rgb* colour system (normalised RGB).

$$r = \frac{R}{R+G+B}\ ,$$

$$g = \frac{G}{R+G+B}\ ,$$

$$b = \frac{B}{R+G+B}$$

Equation 2.10: Normalised RGB colour space

Intensity is factored out so this colour system has the property that its channels are robust to surface orientation, illumination direction and illumination intensity. However, this is dependent on highlights and on the colour of the light source.

2.3.2 Opponent Colour Space

The human visual system is trichromatic, using 3 channels of information to describe an image. The first channel, O1 , is chromatic and is used to identify contrast between red and green colours (positive and negative values respectively). The second channel, O2, is chromatic and is used to identify contrast between yellow and

blue colours. The third channel, O3, is achromatic and is used to identify contrast in intensity (grayscale).

$$O_1 = \frac{R - G}{\sqrt{2}},$$

$$O_2 = \frac{R + G - 2B}{\sqrt{6}},$$

$$O_3 = \frac{R + G + B}{\sqrt{3}}$$

Equation 2.11: Conversion from RGB colour space to Opponent colour space

2.3.3 Opponent SIFT

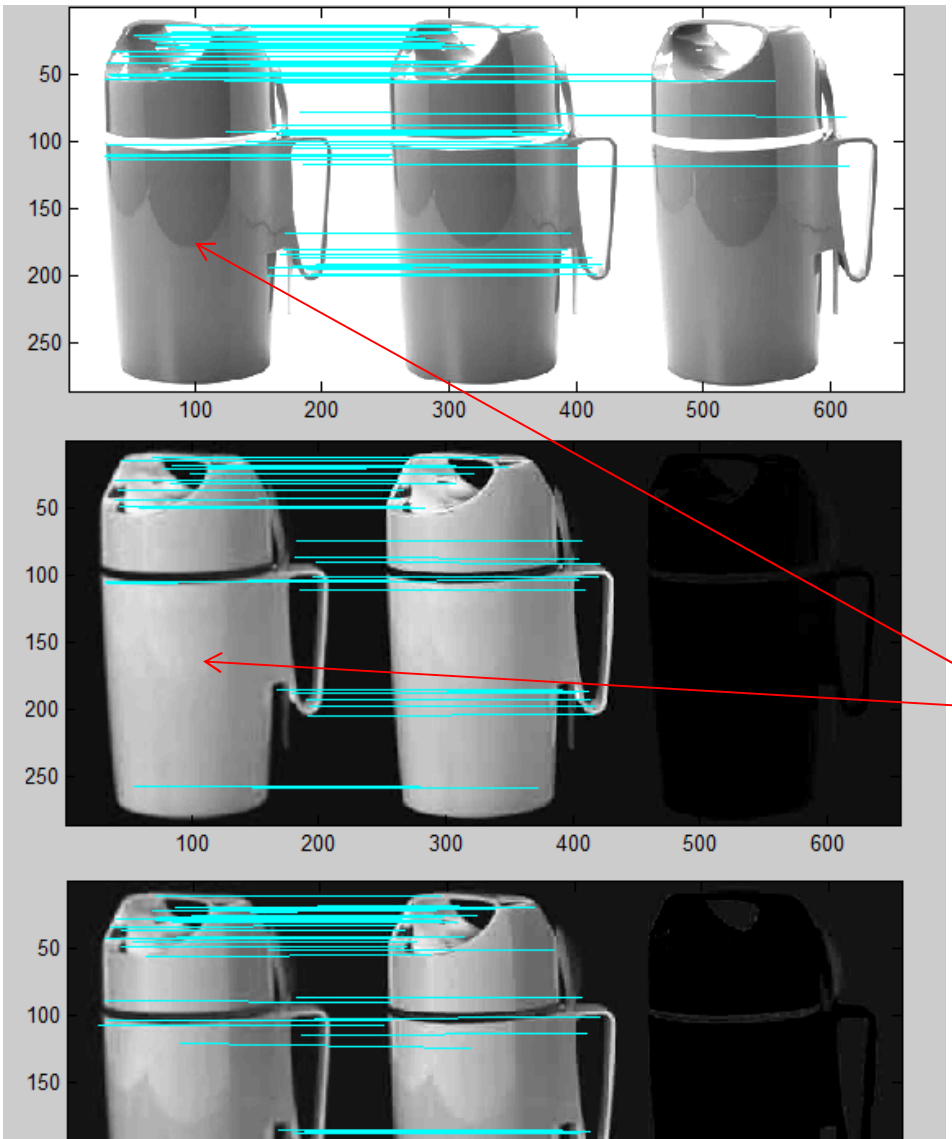
SIFT uses intensity gradients to identify features in an image. Some features may only be identifiable in a chromatic channel (higher contrast). Opponent SIFT computes SIFT descriptors for each channel in Opponent colour space. O3 being achromatic, this would produce the same output as the standard SIFT. O1 and O2 are normalised before SIFT builds a scale-space. The features found in O1 and O2 are features with a contrast in chromaticity. This is useful for identifying targets that may only be distinguishable by their colour.

Below shows an example of how colour descriptors such as opponent SIFT descriptors can be useful.

Figure 2.3: Target

Figure 2.4: Scene

Using this red target the opponent SIFT implementation finds these matches (figure 1.5):



O3, Intensity channel matches

O1, Red-Green channel matches

Invariance to highlights and surface reflectance

O2, Yellow-Blue channel matches

Figure 2.5: SIFT keypoint matches in each channel of Opponent colour space

This example shows how using features from chromatic channels can be useful for determining a more accurate estimate of a specific object. In surveillance, people tracking can be difficult as most people's shapes are the same, and most people have similar achromatic features (at a medium to low resolution). Opponent SIFT can be used in a similar way for surveillance as in the example above, identifying an object by colour and shape, not just shape.

2.4 C++ and Object Orientated Programming techniques

C++ is a cross-platform programming language, used mainly for near real-time computing. With added libraries such as OpenCV and the Qt Framework, it is a suitable language for designing a fast, robust and highly reusable image processing software system.

The prime purpose of C++ programming was to add object orientation to the C programming language, which is one of the most powerful programming languages.

The core of pure object-oriented programming is to create an object, in code, that has certain properties and methods. While designing C++ modules, we try to see a whole world in the form of objects. For example, a car is an object which has certain properties such as colour, number of doors, and the like. It also has certain methods such as accelerate, brake, and so on.

The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

When you define a class, you define a blueprint for a data type. This does not define any data, but defines what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object. A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types.

2.4.1 Inheritance

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.

The idea of inheritance implements the is a relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on. A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

2.4.2 Overloading

C++ allows you to specify more than one definition for a function name or an operator in the same scope, which is called function overloading and operator overloading respectively.

An overloaded declaration is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

When you call an overloaded function or operator, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called overload resolution.

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

You can redefine or overload most of the built-in operators available in C++. Thus a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

2.4.3 Polymorphism

The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance. In C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

A virtual function is a function in a base class that is declared using the keyword virtual. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as dynamic linkage, or late binding. It's possible that you'd want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class, this is called a pure virtual function.

2.4.4 Data Abstraction

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, but you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals.

Now, if we talk in terms of C++ Programming, C++ classes provides great level of data abstraction. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

For example, your program can make a call to the sort() function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work. In C++, we use classes to define our own abstract data types (ADT).

In C++, we use access labels to define the abstract interface to the class. A class may contain zero or more access labels:

- Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.
- Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions.

Data abstraction provides two important advantages:

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data are public, then any function that directly accesses the data members of the old representation might be broken.

Abstraction separates code into interface and implementation. So while designing your component, you must keep interface independent of the implementation so that if you change underlying implementation then interface would remain intact.

In this case whatever programs are using these interfaces, they would not be impacted and would just need a recompilation with the latest implementation.

2.4.5 Encapsulation

All C++ programs are composed of the following two fundamental elements:

- Program statements (code): This is the part of a program that performs actions and they are called functions.
- Program data: The data is the information of the program which affected by the program functions.

Encapsulation is an Object-Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of data hiding.

Data encapsulation is a mechanism of bundling the data, and the functions that use them and data abstraction is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes.

2.4.6 Interfaces (Abstract classes)

An interface describes the behaviour or capabilities of a C++ class without committing to a particular implementation of that class.

The C++ interfaces are implemented using abstract classes and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

A class is made abstract by declaring at least one of its functions as pure virtual function. The purpose of an abstract class (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. Abstract classes

cannot be used to instantiate objects and serves only as an interface. Attempting to instantiate an object of an abstract class causes a compilation error.

Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the ABC. Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error. Classes that can be used to instantiate objects are called concrete classes.

An object-oriented system might use an abstract base class to provide a common and standardized interface appropriate for all the external applications. Then, through inheritance from that abstract base class, derived classes are formed that all operate similarly.

The capabilities (i.e., the public functions) offered by the external applications are provided as pure virtual functions in the abstract base class. The implementations of these pure virtual functions are provided in the derived classes that correspond to the specific types of the application. This architecture also allows new applications to be added to a system easily, even after the system has been defined.

Chapter 3 : THE GEOMETRIC ROTATION OF LINEAR COLOUR-SPACES FOR COLOUR CONSTANCY TRANSFORMATIONS

3.1 Introduction

A novel transformation to improve colour constancy in linear colour spaces is derived. A geometric rotation matrix is used to describe a transformation in opponent colour space; using conventional colour constancy estimators to calculate the angular error between the estimated illuminant vector and the intensity axis (white-light axis in Opponent colour space), defining the angle of rotation for correction. A camera is modelled, based on Planck's law (and so assumes the spectral power distribution follows that of a blackbody) to test the temperature estimation and correction methods. Experiments show that the rotation transformation can improve the colour constancy algorithm performance whilst maintaining a natural image contrast.

A fundamental problem in tracking or object recognition across multiple cameras is colour constancy [25]. Most tracking algorithms use either feature matching or colour histogram analysis (or both) [7], [9], [28]. However, an object in a scene may appear to be of a certain colour depending on three factors during image formation: the properties of the camera being used; the physical properties of the surfaces within a scene; and the angle from which the light source is reflected from a surface (the incident angle) [24], [37], [38], [40]. These factors become problematic in a multiple camera environment due to the many possible different lighting and

background conditions. This becomes even more problematic when a variety of makes and models of cameras are in use.

A simplified model of the intensity measure of a single pixel of an image is described by the following [24], [25], [37], [38], [40]:

$$P_x = \int S(\lambda)E^o(\lambda)Q_x(\lambda)d\lambda \quad (1)$$

Equation 3.1: Simplified model of intensity measure of a single pixel of an image

where S(λ) defines the surface characteristics at a particular spatial location, describing the proportion of light incident at that position that is reflected as a function of wavelength λ. $E^o(\lambda)$ is the spectral power distribution of the scene illuminant and characterizes how much energy the source emits as a function of wavelength. $Q_x(\lambda)$ is the spectral sensitivity function of the imaging device sensor, which determines what proportion of light energy incident upon it is absorbed at each wavelength. Thus the sensor response P_x is a measure of the total energy absorbed by the sensor over the range of wavelengths to which the sensor is sensitive. The subscript x distinguishes a particular colour filter on an imaging sensor (e.g. RG and B for red, green and blue).

The colour constancy problem then can be seen; an object's true colour or spectral response, S(λ), is impossible to recover since it is multiplied by the unknown spectral distribution Eo(λ) . For an un-calibrated camera Qx(λ) is also unknown. Overcoming this problem by recovering S(λ) is clearly valuable since it will result in increased accuracy in recognition and tracking algorithms [23]. There have been many different approaches to colour constancy, each with their own benefits and downfalls [41], [44].The majority of colour constancy algorithms consist of two processes: light source estimation and colour-space transformation [41], [44]. The most successful of the low computationally intensive methods, such as the grey world [41] and grey edge [38] algorithms, use different methods of estimating the light source but employ similar linear colour-space transforms [53] (usually the Von Kries transform [51]) in RGB-space . However, low computationally intensive methods have recently been shown to be able to obtain similar accuracy to more complex, and thus more computationally demanding, methods [37], [38], [41], [44].

The grey-world algorithm assumes the average reflectance in a scene to be achromatic, whereas the grey-edge algorithm assumes the average edge difference in a scene to be achromatic [38]. Finlayson and Trezzi have shown that some of the most popular algorithms (grey-world, max-RGB, shades of grey) can be expressed as the same algorithm but with different Minkowski norms [38], [40].The max-RGB approach is equal to applying the L^{∞} norm and the grey-world is equal to applying the L^{1} norm [38], [40]. A framework for low-level color constancy methods is proposed by Van der Weijer *et al*. which includes the above mentioned algorithms, derived using the framework of Finlayson and Trezzi to include the grey-edge algorithm [38].

Other algorithms use transforms based on data collected over a range of illuminants [39], [45], [54]. Recent color constancy algorithms explore transforms in Opponent color-space [38] or CIE L*A*B* space, due to the preservation of the color saturation in low saturation images [53]. One of the most accurate color constancy algorithms, proposed by Forsyth, is gamut mapping [39], [45], [54]. The gamut of a camera is referred to as the set of all possible RGB values for the illuminant [45]. Gamut mapping produces a map of transforms, usually in RGB space for different illuminants in order to estimate an unknown illuminant. Other accurate approaches to color constancy use probabilistic or learning based methods [54]. These algorithms generally use more computational power or require a large amount of data storage [38].

3.2 Illuminant Temperature Estimation

It has be shown by Yang and Sohn [47] that being able to estimate the temperature of an image can dramatically increase the accuracy of color constancy algorithms. These describe how the image formation model can be manipulated to estimate image temperature. The spectral sensitivities of the camera sensors can be approximated using delta functions such that.

$$Q_x(\lambda) = q_x \delta(\lambda - \lambda_x), x \in \{R, G, B\}$$

(2)

where λx defines the single sensitive wavelength of each sensor.

The temperature of a light source can be calculated using Planks law with the assumption that the spectral power distribution of light follows that of a black body (using the term 'A' as a scaling factor):

$$E_{(\lambda)} = A \frac{2hc^2}{\lambda^5} \frac{1}{\frac{hc}{e\lambda kT} - 1}$$

(3)

Yang and Sohn simplify the formula as follows [47]:

$$c_1 = 2hc^2$$
$$c_2 = hc \,/\, k$$

so:

$$E(\lambda) = A \frac{c_1}{\lambda^5} \frac{1}{\frac{c_2}{e\lambda T} - 1}$$

$$E(\lambda) \approx A c_1 \lambda^{-5} e^{-\frac{c_2}{\lambda T}}$$

(4)

The image acquisition process can therefore be expressed as [47]:

$$p_x^i = \sigma A c_1 \lambda_x^{-5} e^{-\frac{c_2}{\lambda_x T}} S^i(\lambda_x) q_x \,, \; x \in \{R, G, B\}$$

(5)

Yang and Sohn simplify the expression further to extract temperature 'T' so that [47]:

$$\frac{1}{T} = \frac{1}{C_2} \left(\frac{\lambda_x \lambda_G}{\lambda_x - \lambda_G} \right) \times ln \left(\frac{\lambda_G^{-5} q_G E[p_x^i]}{\lambda_x^{-5} q_x E[p_G^i]} \right),$$

$$x \in \{R, B\}$$

(6)

This method of estimating the color temperature is shown to improve the illuminant estimate as compared with statistical approaches.

3.3 Working with Opponent Color Space

The human visual system is trichromatic, using 3 channels of information to describe an image [38]. Opponent color-spaces, for example the CIE L*a*b* space, mimic this color system by using two channels for chromatic information and one channel for intensity (or luminance as is the case in L*a*b* space). The first channel in Opponent space, referred to in this case as O1, is chromatic and is used to identify contrast between red and green colors (positive and negative values respectively). The second channel, O2, is chromatic and is used to identify contrast between yellow and blue colors. The third channel, O3, is achromatic and is used to identify contrast in intensity (grey-scale). Opponent space has similar properties to normalized RGB space with the O1 and O2 channels invariant to intensity [38].

Intensity invariant values can be obtained by normalizing RGB values by their intensity (Intensity = R+G+B) resulting in the *rgb* color system (normalized RGB).

$$r = \frac{R}{R+G+B}\ ,$$
$$g = \frac{G}{R+G+B}\ ,$$
$$b = \frac{B}{R+G+B}$$

(7)

This color system has the property that its channels are robust to surface orientation,

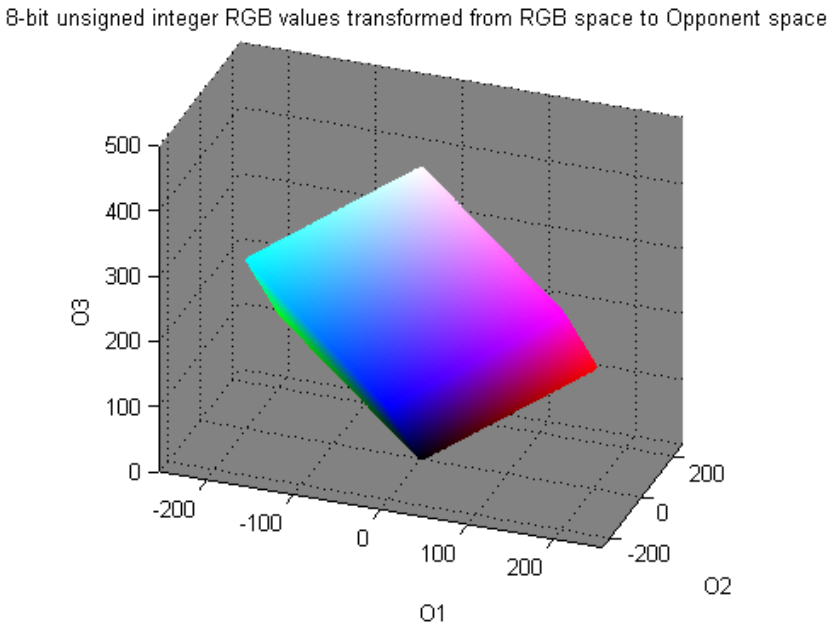


Figure 3.1: RGB surface values of a cube transformed into Opponent space.

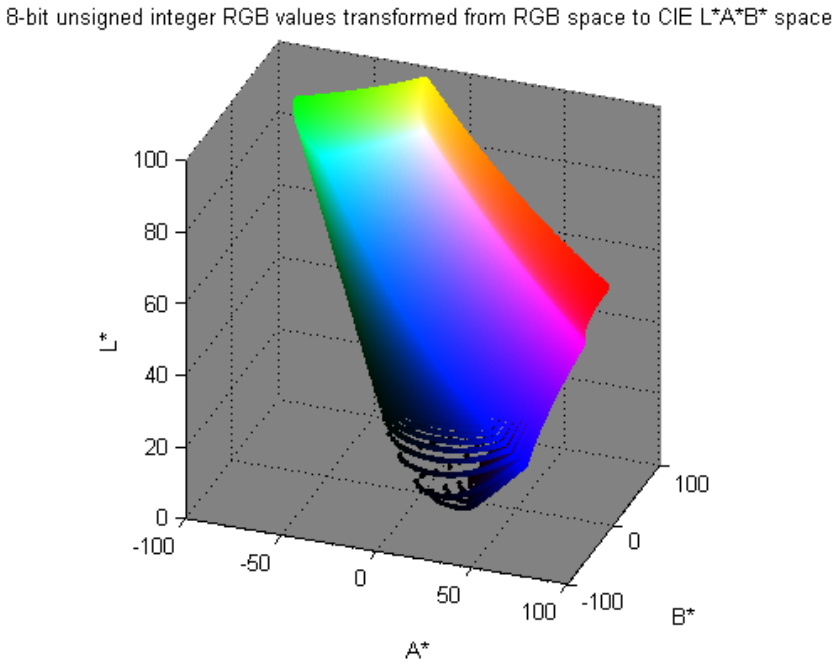


Figure 3.2: RGB surface values of a cube transformed into CIE L*A*B* space

highlights and the color of the light source [6]. Figure 1 shows visually how the

RGB color space is transformed into Opponent color space.

CIE L*a*b* space works on the basis that a change in color perception correlates to an equal change in color space, based on the D65 illuminant (commonly used standard illuminant defined by the International Commission on Illumination (CIE)) which corresponds roughly to a midday sun in Western Europe / Northern Europe [28]. As humans are more sensitive to green light, the green areas in L*a*b* space correlate almost linearly with RGB space, whereas the blue and red areas are stretched out (or curved). This curving effect can be seen in Figure 2, showing RGB values converted to CIE L*a*b*.

3.4 Camera Simulation

A simulation of a Canon D60 DSLR, based on the image acquisition model [9][4][7][8] and temperature estimation model [19], was used to estimate how a black-body surface average reflection changes in opponent-space depending on illuminant temperature, using the following camera characteristics described below:

Approximate Frequency Responses of Canon D60 DSLR [27]:

Red frequency response = 600 nm
Green frequency response = 530 nm
Blue frequency response = 450 nm

The camera simulation models a photodiode of a digital single-lens reflex camera by normalizing the RGB channels by the maximum illuminant intensity across all wavelengths. Figure 3 shows the illuminant at different temperatures in Opponent space.

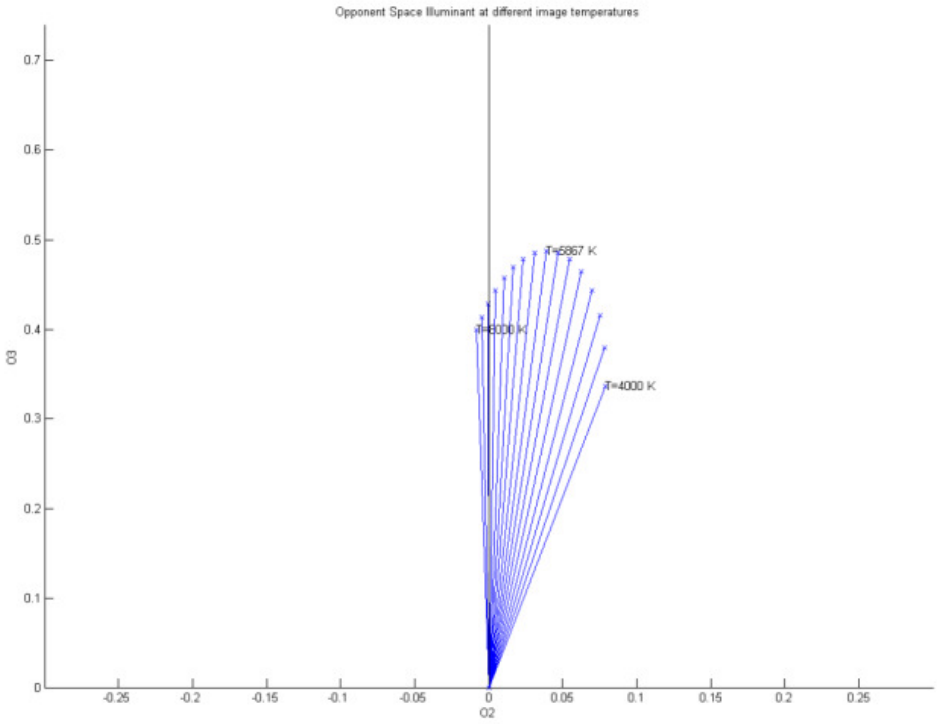


Figure 3.3: The simulation of an illuminant at different temperatures in Opponent space

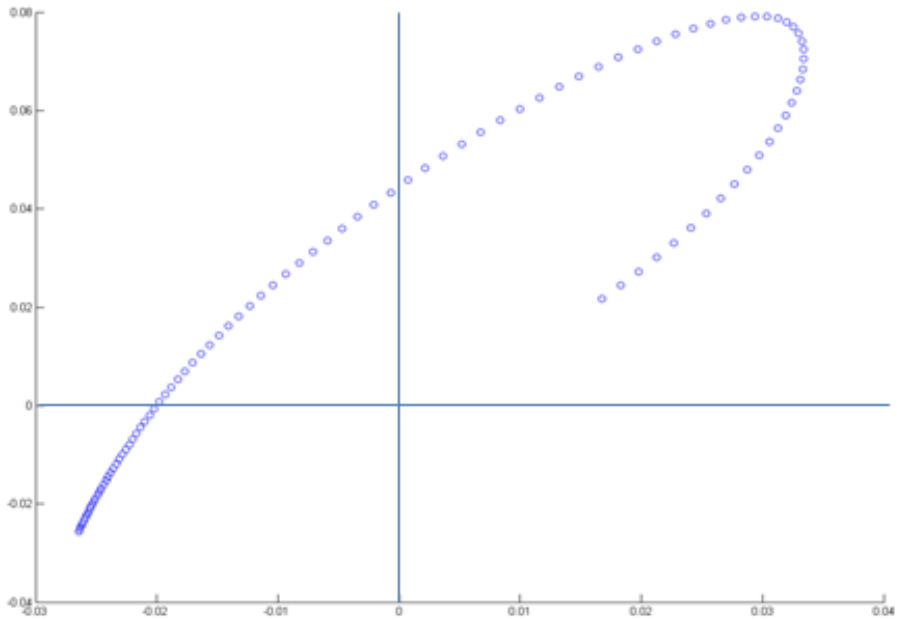


Figure 3.4: The simulation of an illuminant at different temperatures in opponent space, channels O1 and O2 (chromatic) only.

Figure 3 shows how the illuminant color changes non-linearly in Opponent space as the illuminant temperature increments linearly. If we ignore the intensity (O3 channel [6]) and plot only the chromatic information, the non-linear movement becomes clearer. This can be seen in the figure 4.

These graphs show that the illuminant moves through the color-space non-linearly, more similar to a rotation than a skew [6][25]. The illuminants change in position in color-space would be considered equal when using a skew (or at least change position at a similar rate).

This rotation effect does not occur in L*a*b* space due to it being based on the D65 illuminant. The areas in which the illuminant moves slowly through the Opponent color space (red and blue, low and high temperature illuminants respectively) move with equal distance in L*a*b* space. This “curving” effect can be seen in figure 5, where a linear square of L*a*b* values is transformed into Opponent space (figure 6).

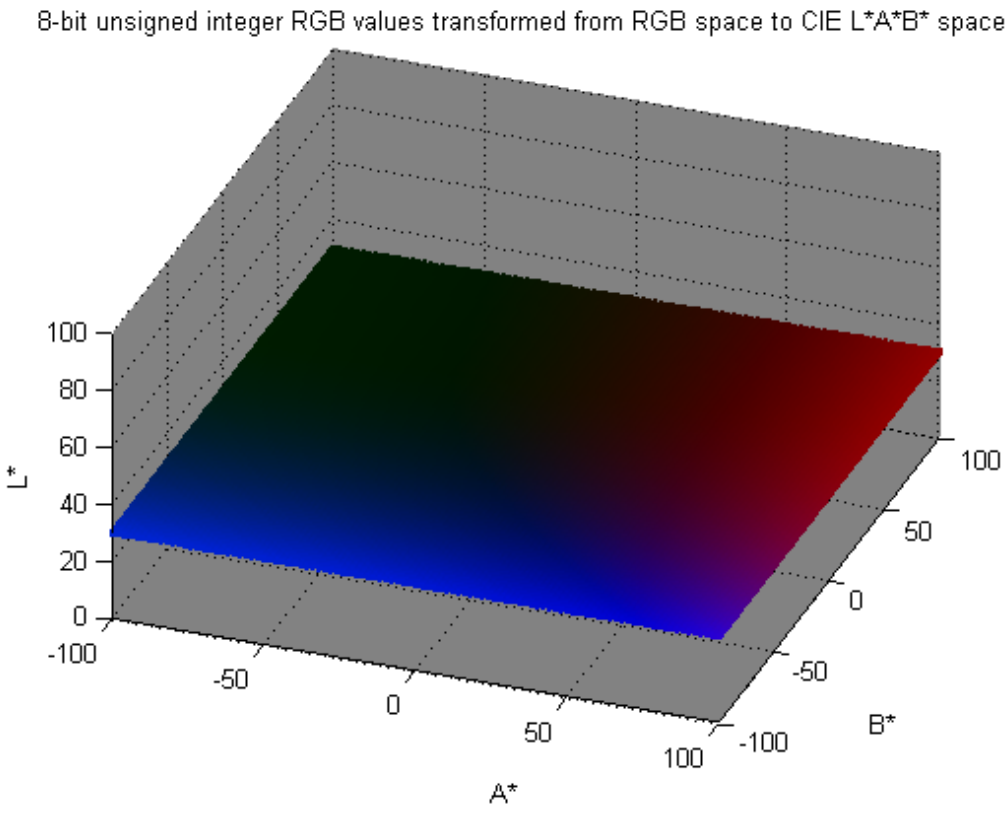


Figure 3.5: L*a*b* square, with Luminance 30

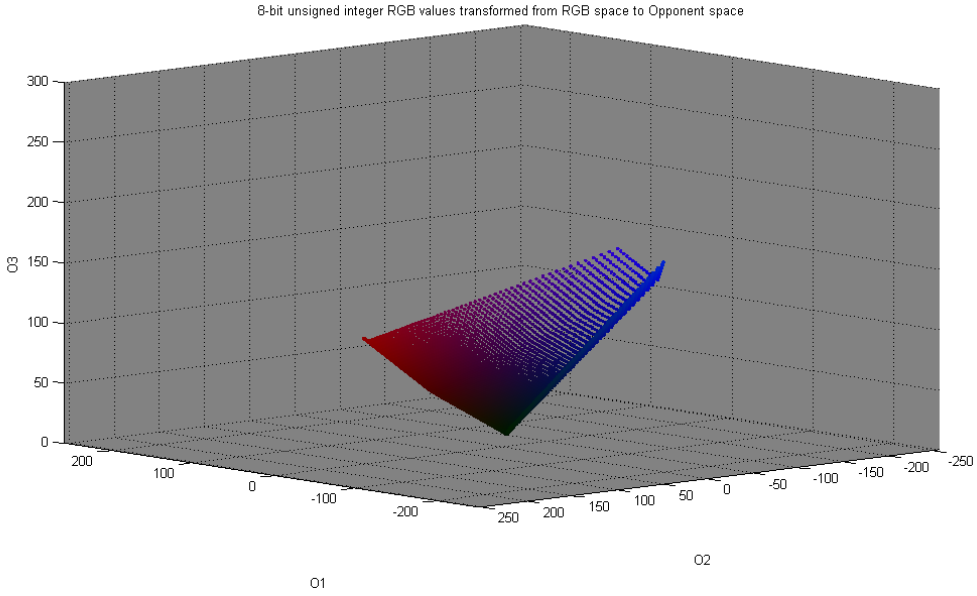


Figure 3.6: L*a*b* square from figure 5 transformed into Opponent space

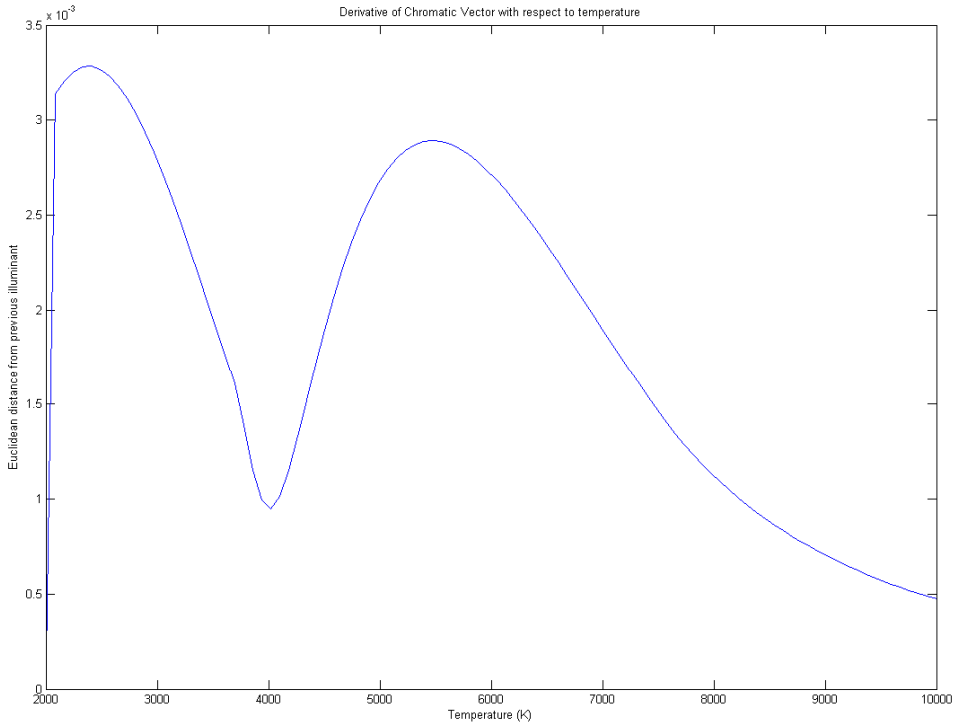


Figure 3.7: The differentiation of O1 and O2 channels with respect to illuminant temperature

A rotation describes the rate of change in illuminant position more accurately as it takes into account the acceleration and deceleration of illuminant position at the high and low ends of the temperature spectrum. Due to different camera characteristics the observed curve may be slightly elliptical or skewed towards a specific color but it can be modeled using a rotation matrix. Figure 7 shows the differentiation of the illuminant's position in the color space with respect to temperature. The differentiation of the illuminant position shows that it moves nonlinearly. Between 4000K and 8000K, the Euclidean distance from the illuminant's previous position starts off low and then increases until it reaches the "white point" after which it then declines.

3.5 Rotational Transformation

In this section we pursue color constancy by finding the average geometric vector in Opponent space, assuming the average light in a scene to be achromatic, similar to the Grey World and Grey-Edge hypotheses [6]. This method is based on the observation, by Joost van de Weijer *et al*, that the distribution of colour derivatives exhibit the largest variation in the light source direction [6]. The rotational transform can be used with other standard color constancy algorithms, using the illuminant estimate as input.

The colour-space is first converted from RGB to Opponent color-space to gain intensity invariant values as follows:

$$O1 = \frac{R-G}{\sqrt{2}},\ O2 = \frac{R+G-2B}{\sqrt{6}},\ O3 = \frac{R+G+B}{\sqrt{3}} \quad (8)$$

To estimate the illuminant vector direction in opponent space we find the average angle α of the chromatic vectors $[O1, O2]^T$, however other estimators can be used.

$$\alpha = tan^{-1}\frac{\sum_{i=1}^{n} sin\,(tan^{-1}\frac{O1_i}{O2_i})}{\sum_{i=1}^{n} cos\,(tan^{-1}\frac{O1_i}{O2_i})}$$

Equation 3.2: Average angle of the chromatic vectors in Opponent colour space

(9)

The rotation axis vector u, which is perpendicular to the average direction of the chromatic vectors $[O1, O2]^T$ can then be expressed as:

$$u\begin{bmatrix}x\\y\\z\end{bmatrix} = \begin{bmatrix}\sin(\frac{\pi}{2} + \tan^{-1}\frac{\sum_{i=1}^{n}\sin(\tan^{-1}\frac{O1_i}{O2_i})}{\sum_{i=1}^{n}\cos(\tan^{-1}\frac{O1_i}{O2_i})})\\ \cos(\frac{\pi}{2} + \tan^{-1}\frac{\sum_{i=1}^{n}\sin(\tan^{-1}\frac{O1_i}{O2_i})}{\sum_{i=1}^{n}\cos(\tan^{-1}\frac{O1_i}{O2_i})})\\ 0\end{bmatrix}$$

Equation 3.3: Rotation axis vector calculation

(10)

The angle of rotation θ towards the white light axis is calculated as the average angle between the vectors $[O1, O2]^T$ (or pre-estimated illuminant vector) and the white-light axis (O3, or z-axis):

$$\theta = -\tan^{-1}\frac{\sum_{i=1}^{n}\sin(\tan^{-1}\frac{O3_i}{\sqrt{O1_i^2 + O2_i^2 + O3_i^2}})}{\sum_{i=1}^{n}\cos(\tan^{-1}\frac{O3_i}{\sqrt{O1_i^2 + O2_i^2 + O3_i^2}})}$$

Equation 3.4: Rotation angle towards the white light axis (error angle)

(11)

The rotation matrix R with rotation θ around axis u can then be expressed as:

$$R\begin{bmatrix}O1\\O2\\O3\end{bmatrix} = \begin{bmatrix}\cos\theta + u_x^2(1-\cos\theta) & u_xu_y(1-\cos\theta) - u_z\sin\theta & u_xu_z(1-\cos\theta) + u_y\sin\theta\\ u_yu_x(1-\cos\theta) + u_z\sin\theta & \cos\theta + u_y^2(1-\cos\theta) & u_xu_z(1-\cos\theta) - u_x\sin\theta\\ u_zu_x(1-\cos\theta) - u_y\sin\theta & u_zu_y(1-\cos\theta) + u_x\sin\theta & \cos\theta + u_z^2(1-\cos\theta)\end{bmatrix}\cdot\begin{bmatrix}O1\\O2\\O3\end{bmatrix}$$

Equation 3.5: Rotation matrix for correcting chomatic average vector

(12)

After rotation, the color-space is converted back to RGB:

$$R = O1\sqrt{2} + G$$

$$G = \frac{O3\sqrt{3} - B - O1\sqrt{2}}{2}$$

$$B = \frac{O3\sqrt{3} - O2\sqrt{6}}{3}$$

Equation 3.6: Opponent colour space to RGB colour space conversion

(13)

For the purposes of color constancy, the O3 channel can be left unchanged as it contains intensity information only [6].

3.6 Experiments

The rotational transform was compared to conventional color constancy transforms at different temperatures. A Canon D60 DSLR was used to capture images of an "X-

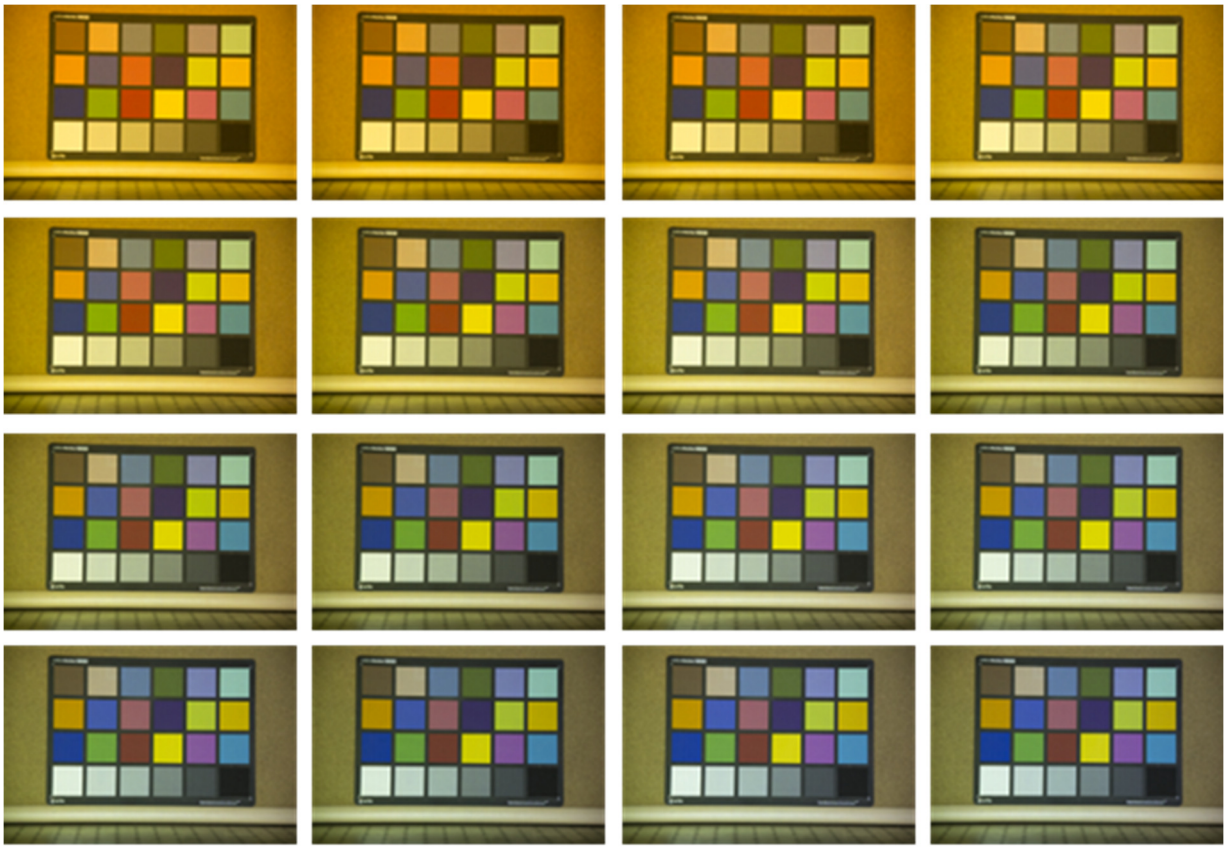


Figure 3.8: X-Rite color checker board at different temperatures

Rite" GretagMacbeth® color checker board at 16 evenly distributed illuminant temperatures, projected from a calibrated light source. The illuminants were generated using the camera simulation model, with temperatures ranging from approximately 4000K to 8000K (approximated from Yang and Sohns temperature estimation method). Automatic white balance and temperature settings were disengaged on the camera to ensure no color constancy methods were applied to the raw images before testing.

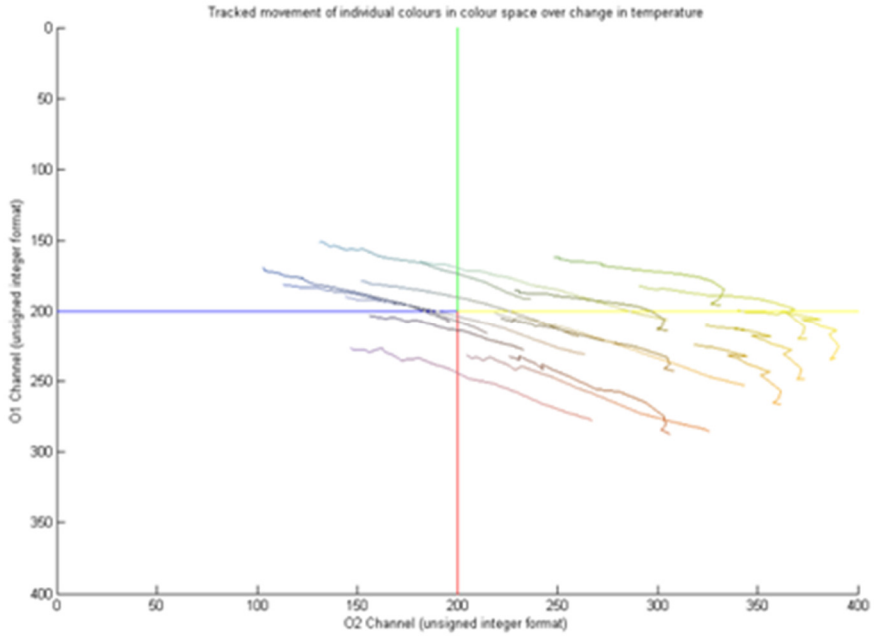


Figure 3.9 Tracked movement of each colour from the colour checker board over different temperatures.

The Rotational Transform was applied to the captured images using conventional colour constancy algorithm estimators (such as Grey-World, Shades of Grey etc.) and compared to the outputs of conventional colour constancy algorithms that use linear transform methods (i.e. Von Kries). From the experiments we find the median angular error between each algorithms own outputs, as well as testing algorithms outputs against each other (some systems may use different algorithms at different temperatures for optimization).

Figure 8 shows an example of some of the test images used with different temperature light sources for each. Figure 9 shows how each color from the color checker board changes with temperature.

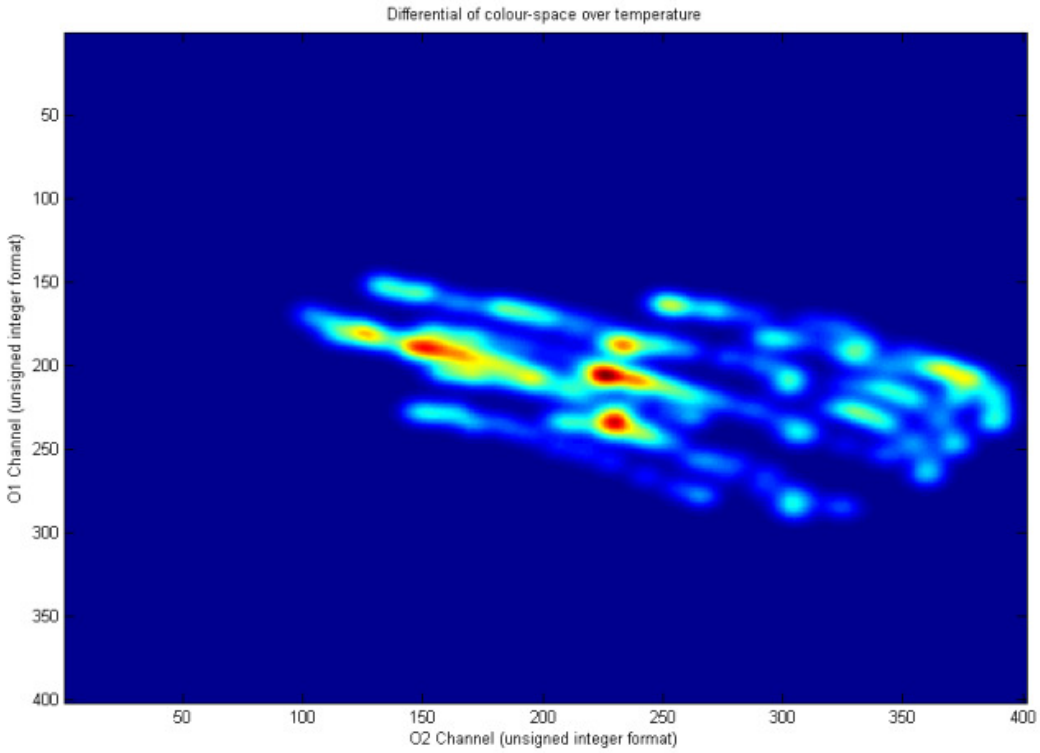


Figure 3.10: Density of tracked color patches over different temperatures

The density of the colourspace with respect to temperature is calculated using a 3 dimensional bitmap, 2 being spacial (chromatic space) and the 3rd being temperature. After smoothing the bitmap with a gaussian convolution, the image shown in figure 10 is formed. The dark red areas represent a small amount of colour change (positions at different temperatures close together in colourspace). The light blue areas are areas where the colours changed fast as the temperature increased. Each individual colour from the colour checker board moves differently depending on the temperature, this can be seen in figure 11.

The angular error between pixels of an image is calculated as follows:

$$a = \sqrt{R1^2 + G1^2 + B1^2}$$

$$b = \sqrt{R2^2 + G2^2 + B2^2}$$

$$e = \sqrt{(R1 - R2)^2 + (G1 - G2)^2 + (B1 - B2)^2}$$

$$\theta_i = cos^{-1} \frac{a_i^2 + b_i^2 - e_i^2}{2a_i b_i}$$

Equation 3.7: Calculation of angular error between pixels of an image in RGB

where a is the RGB pixel vector on the target image, b is the RGB pixel vector on the test image, e is the error vector, θ is the angular error and i is the pixel number.

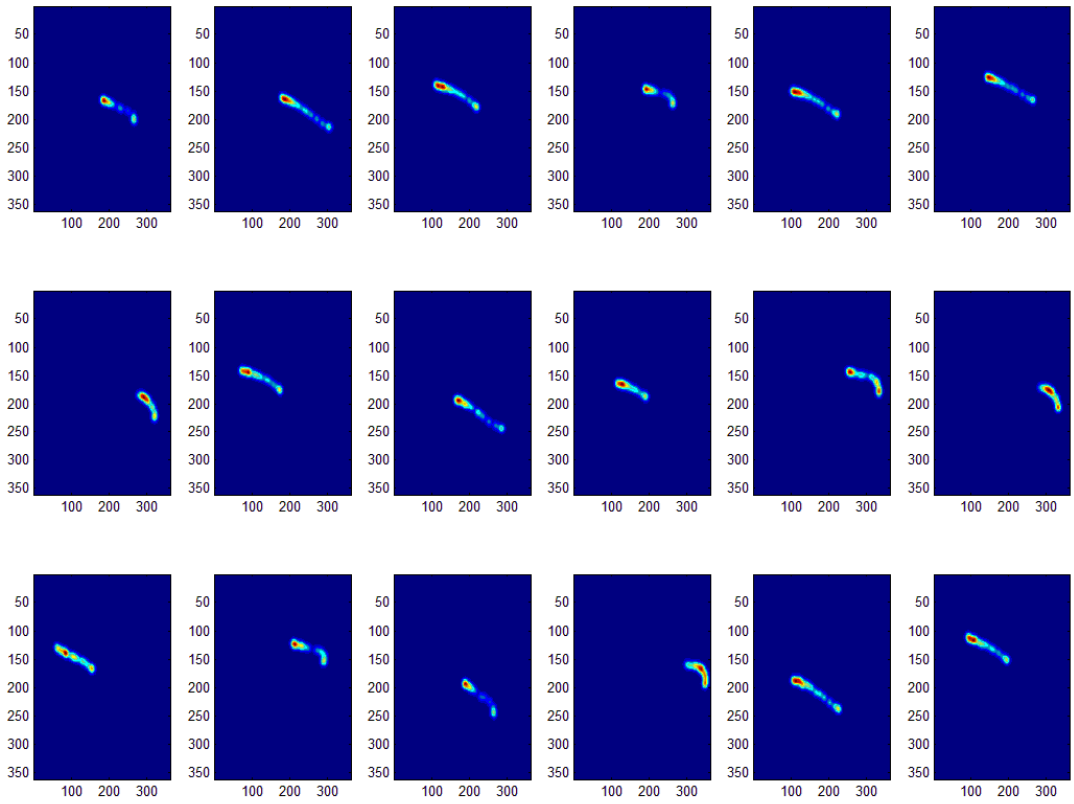


Figure 3.11: Individual colour movements at different temperatures in Opponent colour space

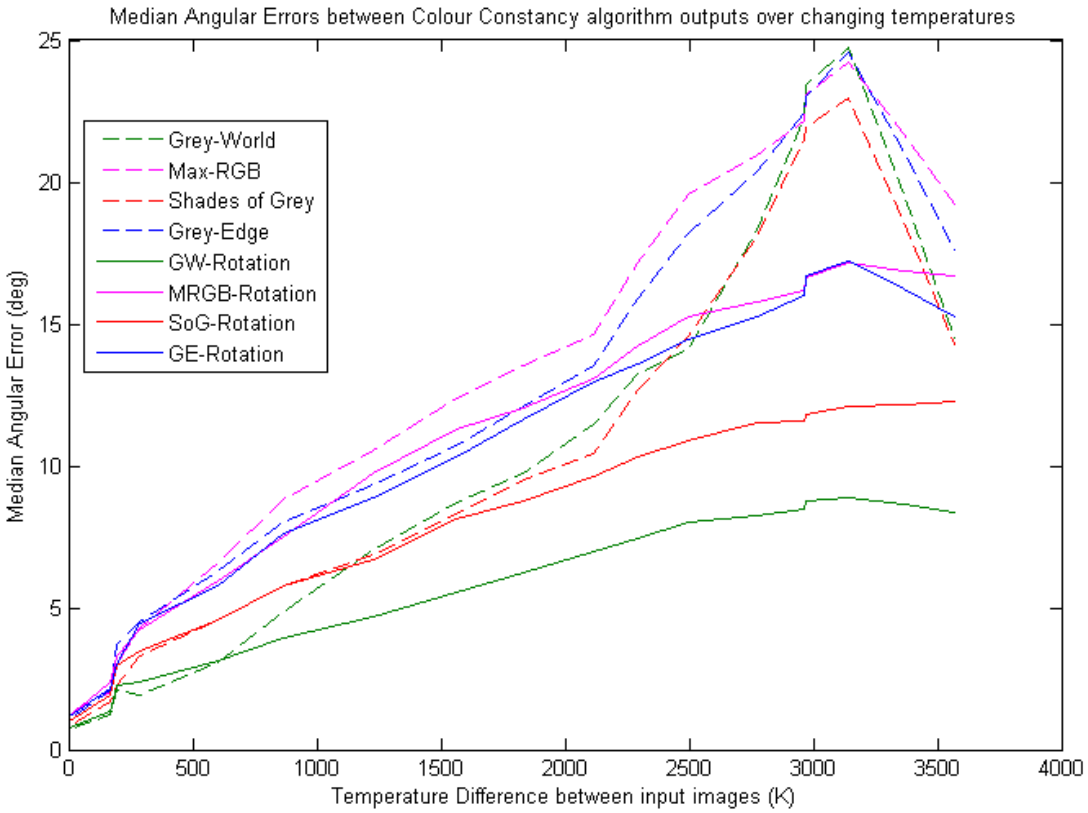


Figure 3.12: Median angular errors (degrees) for tested color constancy algorithms over changing temperatures (K)

Figure 12 shows the median angular error of each algorithms output between varying temperatures. The dotted lines show the original algorithms errors and the single lines show the algorithms errors using a rotation transform instead of linear.

Algorithm (over all temperatures)	Median of Angular Errors with a linear transform (3d.p.)	Median of Angular Errors with a rotational transform (3d.p.)	Accuracy increase (% 3d.p.)
Grey-World	10.617	6.613	37.710
Max-RGB	14.066	12.550	10.777
Shades of Grey	9.984	9.201	7.843

Grey-Edge	12.827	12.303	4.089

Table 1: Angular errors from tested algorithms (tested against their own outputs)

The algorithms outputs were tested against each other's outputs at different temperatures to find the median angular error. The error would show how accurate a system would be if multiple algorithms were in use across multiple cameras. Table 2 shows the results of the algorithms tested.

Figure 15 shows the color depth for each color constancy output at different temperatures. The color depth is calculated as the number of unique Hue values in an image (with 256 quantization levels). The color depth of the input image increases as it approaches higher image temperatures as expected. The outputs from the color constancy algorithms should be at a constant color depth (gamut mapping chooses the gamut with the largest color depth [25]). The algorithm with the lowest angular error (Grey-World) also has the most constant color depth across all temperatures.

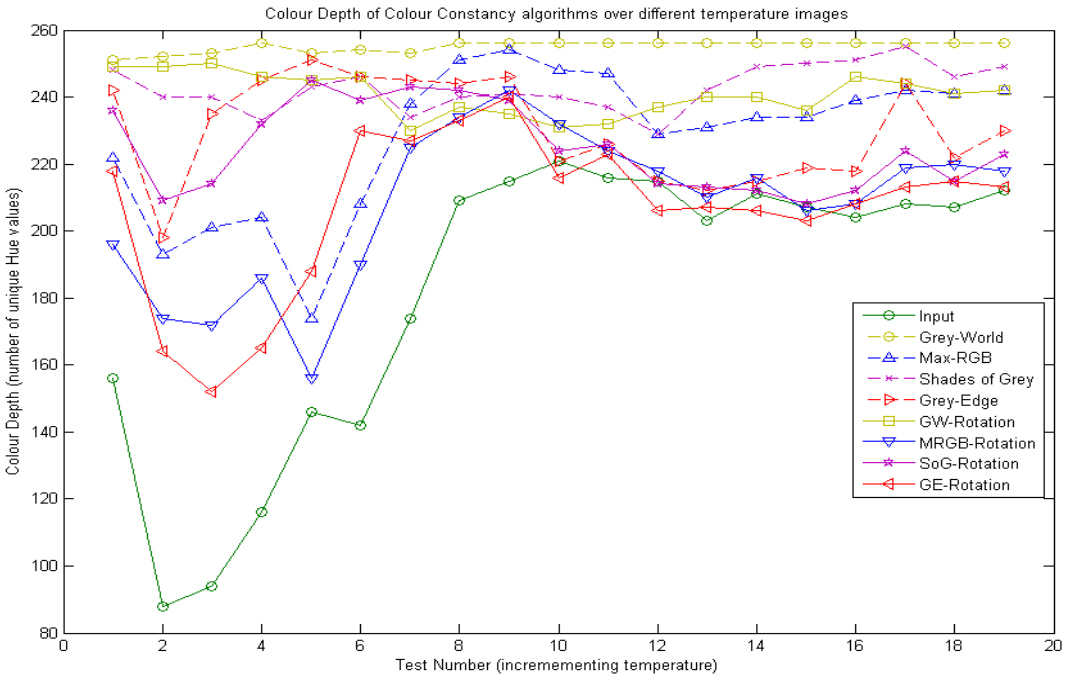


Figure 3.13: Color depth of each algorithms output compared to the input images (at different temperatures)

Table 2: Angular errors from tested algorithms (tested against each other's outputs)

						Rotational Transform			
Linear Transform		Grey-World	Max-RGB	Shades of Grey	Grey-Edge	Grey-World	Max-RGB	Shades of Grey	Grey-Edge
	Grey-World	10.617	15.775	15.266	15.642	14.697	16.358	16.448	16.358
	Max-RGB		14.066	15.428	14.455	18.308	13.847	14.748	14.202
	Shades of Grey			9.984	8.859	12.552	8.287	9.079	8.542
	Grey-Edge				12.827	16.457	12.308	13.092	12.57
Rotational Transform	Grey-World					6.613	4.413	4.016	4.48
	Max-RGB						12.55	14.171	13.307
	Shades of Grey							9.201	8.337
	Grey-Edge								12.303

The images in figures 16 and 17 show the input and output of the Grey-World algorithm using the Rotational Transform on a real-world scene. Figures 18 and 19 show the pixel density in opponent color space. The corrected image (figure 17) has a more balanced color space, with a highly dense column of colors passing through the O3 (white light) axis.

Figure 3.15 Input image without white balance

Figure 3.14 Output of Grey-world rotational

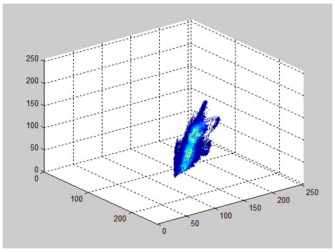


Figure 3.17 Opponent colour space density of input image

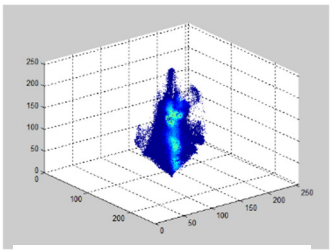


Figure 3.16: Opponent colour space density of output image

Figure 3.18 comparison of tested colour constancy algorithms outputs for a low temperature image, using the Shades of Grey algorithm as illuminant estimate for rotational transform (see equation 9)

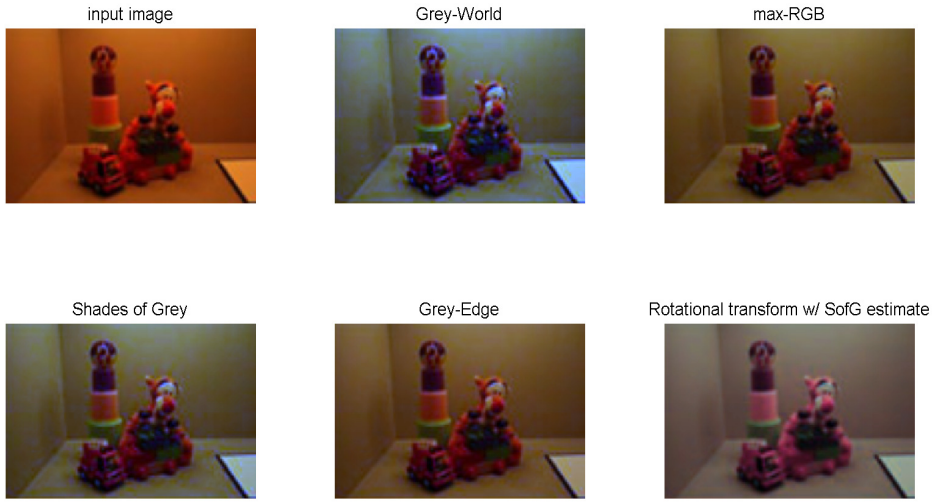


Figure 3.19 A comparison of tested color constancy algorithms outputs for images from Yang and Sohn's "Image-based color temperature estimation for color constancy" paper [19], showing similar results in accuracy but with unknown camera characteristics

3.7 Conclusions

The tests conducted show that the outputs of different color constancy algorithms change dramatically when the images being processed have different light source temperatures. This was expected as Barnard et al. [10][13] and many others document this effect. However, the rotational transform, based on the DSLR camera simulation, has been shown to reduce the median angular error between outputs at different temperatures.

The grey-world algorithm was most improved (by 37.7%) and performed best at high temperature differences. The grey-edge algorithm performed best on images that had very little temperature difference. The rotational transform yields the least error at higher temperature changes compared to a linear transform which yields least error between images with lower temperature changes. Images with approximate temperatures in the range of 5500K and 6500K appear to contain more green light reflected from the surfaces within the scene than images with extreme high or low temperature estimates. In CIE L*a*b* color-space, the green areas color

resolution remains the same as in RGB space, however the red and blue areas appear to be of a higher resolution than RGB space, creating the curving effect seen in figure 2. This curving effect is the reason why linear transforms perform better on the mid-range temperature images compared to the rotational transform, and vice versa.

Further experimentation with the algorithms tested against each other in cases of hybrid cross-algorithm systems, showed that the rotational transform yielded the least median error between the other algorithms as compared to the standard transforms, with 4 degrees median angular error between the grey-world and shades of grey estimators. The rotational transform also maintains image contrast as compared to other algorithms; this can be seen visually in the images shown in Figure 20. This is mainly due to the RGB and Opponent space color perception errors. Chong *et al.* [23] states how transformations in CIE L*a*b* space produce "more natural" images due to it being similar to the human visual system.

Maintaining color contrast is important in cases of color feature tracking. Opponent-SIFT [6] detects color features based on a Difference of Gaussian pyramid in the chromatic channels of Opponent space. The "false edges" created by some algorithms outputs (grey-world and shades of grey in Figure 20) would disrupt such feature detection algorithms.

In conclusion, the rotational transform in Opponent space provides a noticeable difference both statistically and visually, decreasing angular error of color constancy outputs (based on their illuminant estimations) whilst maintaining color contrast. In a real-word scenario the rotational transform can be implemented into a color tracking system with illuminant estimators to improve feature detection under low temperature lighting conditions.

Chapter 4 : IMPROVING MULTIPLE CAMERA COLOUR CONSTANCY PERFORMANCE WITH GENETIC PROGRAMMING

4.1 Introduction to Genetic Programming

Genetic programming is an evolutionary computation technique that can be used to solve problems automatically, starting from a high-level statement or input of data that needs to be fit to [55]. This technique mirrors aspects of evolution in nature by using the "survival of the fittest" approach to finding a solution. It has been used to solve a wide range of practical problems, producing a number of human-competitive solutions and even new inventions [55]–[60].

Genetic programming is commonly described as evolving programs. However, instead of using Turing-complete languages used in software development, it is more appropriate to evolve programs in a constrained, domain-specific language [55], [57], [61]. Such a language is formed from the definition of the terminal and function sets. A terminal set can be either a programs external inputs (variables) or constants. The function set of the genetic program is normally chosen depending on the nature of the problem domain. A function set could consist of arithmetic functions (e.g. +, -, *, /), Boolean functions (e.g. AND, OR) or specialised functions specifically designed for the problem domain. It is common to require that all functions be type consistent. Crossover breeding might explicitly make use of type information so that the children they produce do not contain illegal type mismatches [55], [61], [62].

A genetic programming system works by creating a random initial set of algorithms in tree structures. Each node of the tree can either be a variable, constant or operator (function) [55]. The number of programs contained in the system is referred to as the population [55]. Each program is executed and evaluated using a fitness case. The fittest programs (programs that produce the least error) are selected for crossover breeding [55]. Crossover breeding is a method of combining two algorithms to create

a new, fitter algorithm. It is achieved by creating copies of the parent trees, selecting a random node from each and then joining them at that node [55], [62].

This type of system has been used in computer vision to select which colour constancy algorithm to use in different lighting conditions from image analysis [58]. It has also been applied to 2D gamut mapping with a human evaluator, selecting the fittest programs manually from the appearance of a programs output [60]. However, it has not been used at a more abstract level, generating new colour constancy algorithms without human intervention. Such a system would be beneficial in a large multi-camera environment where human intervention or setup would be costly.

4.1.1 Representation

Genetic programs are usually represented as tree structures, with each leaf of the tree either being a variable, operator function or a constant. Variables and constants are called terminals, where as operator functions (arithmetic operations or other special functions) are internal nodes, as they require at least one input node.

In genetic programming, programs are usually expressed as syntax trees rather than as lines of code. For example, the variables and constants in the program (x, y and 3) are leaves of the tree. In genetic programming they are called terminals, whilst the arithmetic operations (+, * and max) are internal nodes called functions. The sets of allowed functions and terminals together form the primitive set of a genetic programming system.

In more advanced forms of genetic programming, programs can be composed of multiple components (e.g., subroutines). In this case the representation used in genetic programming is a set of trees (one for each component) grouped together under a special root node that acts as glue, as illustrated in Figure 2.2. We will call these (sub)trees branches. The number and type of the branches in a program,

together with certain other features of their structure, form the architecture of the program.

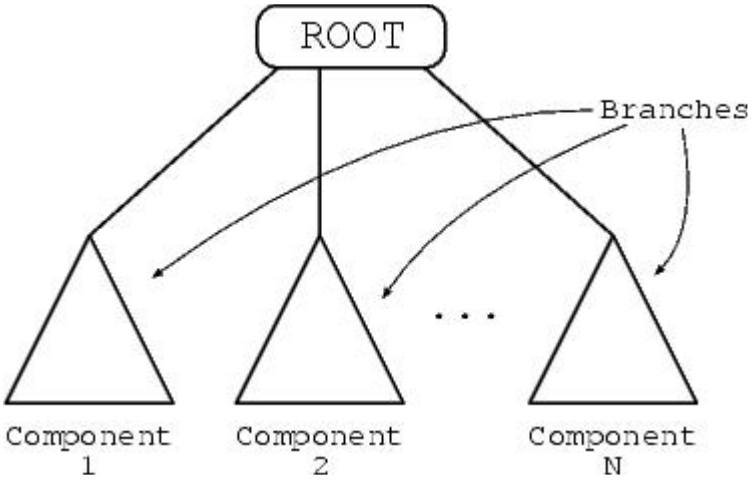


Figure 4.1 Multi-component program representation.

It is common in the genetic programming literature to represent expressions in a prefix notation. For example, max(x+x,x+3*y) becomes (max (+ x x) (+ x (* 3 y))). This notation often makes it easier to see the relationship between (sub)expressions and their corresponding (sub)trees. Therefore, in the following, we will use trees and their corresponding prefix-notation expressions interchangeably.

How one implements genetic program trees will obviously depend a great deal on the programming languages and libraries being used. Languages that provide automatic garbage collection and dynamic lists as fundamental data types make it easier to implement expression trees and the necessary genetic programming operations. Most traditional languages used in artificial intelligence research, many recent languages (e.g., Ruby and Python), and the languages associated with several scientific programming tools (e.g., MATLAB and Mathematica) have these facilities. In other languages, one may have to implement lists/trees or use libraries that provide such data structures.

In high performance environments, the tree-based representation of programs may be too inefficient since it requires the storage and management of numerous pointers. In some cases, it may be desirable to use genetic program primitives which accept a

variable number of arguments. However, fortunately, it is now extremely common in genetic programming applications for all functions to have a fixed number of arguments. If this is the case, then, the brackets in prefix-notation expressions are redundant, and trees can efficiently be represented as simple linear sequences. In effect, the function's name gives its arity and from the arities the brackets can be inferred. For example, the expression (max (+ x x) (+ x (* 3 y))) could be written unambiguously as the sequence max + x x + x * 3 y.

The choice of whether to use such a linear representation or an explicit tree representation is typically guided by questions of convenience, efficiency, the genetic operations being used (some may be more easily or more efficiently implemented in one representation), and other data one may wish to collect during runs. (It is sometimes useful to attach additional information to nodes, which may be easier to implement if they are explicitly represented).

These tree representations are the most common in genetic programming, e.g., numerous high-quality, freely available genetic program implementations use them.

4.1.2 Initialising the population

Like in other evolutionary algorithms, in genetic programming the individuals in the initial population are typically randomly generated. There are a number of different approaches to generating this random initial population. Here we will describe two of the simplest (and earliest) methods (the full and grow methods), and a widely used combination of the two known as Ramped half-and-half.

In both the full and grow methods, the initial individuals are generated so that they do not exceed a user specified maximum depth. The depth of a node is the number of edges that need to be traversed to reach the node starting from the tree's root node (which is assumed to be at depth 0). The depth of a tree is the depth of its deepest leaf . In the full method (so named because it generates full trees, i.e. all leaves are at the same depth) nodes are taken at random from the function set until the maximum tree depth is reached. (Beyond that depth, only terminals can be chosen.) Figure 2.3

shows a series of snapshots of the construction of a full tree of depth 2. The children of the * and / nodes must be leaves or otherwise the tree would be too deep. Thus, at both steps t = 3, t = 4, t = 6 and t = 7 a terminal must be chosen (x, y, 1 and 0, respectively).

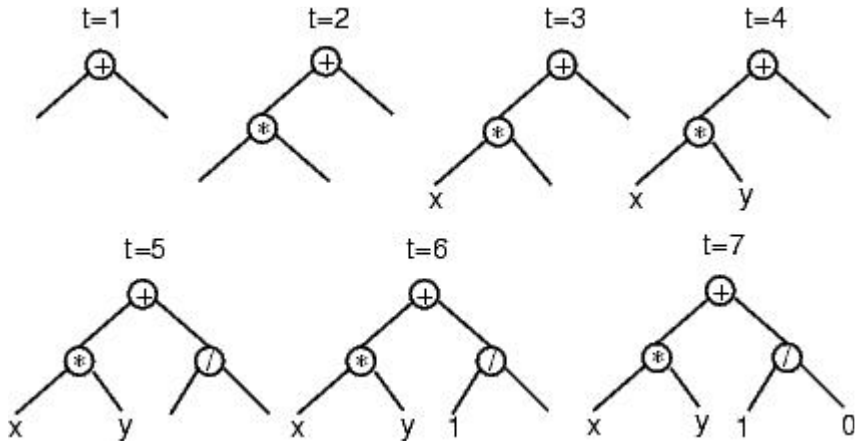


Figure 4.2 Creation of a full tree having maximum depth 2 using the full initialisation method (t = time).

Although, the full method generates trees where all the leaves are at the same depth, this does not necessarily mean that all initial trees will have an identical number of nodes (often referred to as the size of a tree) or the same shape. This only happens, in fact, when all the functions in the primitive set have an equal arity. Nonetheless, even when mixed-arity primitive sets are used, the range of program sizes and shapes produced by the full method may be rather limited. The grow method, on the contrary, allows for the creation of trees of more varied sizes and shapes. Nodes are selected from the whole primitive set (i.e., functions and terminals) until the depth limit is reached. Once the depth limit is reached only terminals may be chosen (just as in the full method). Figure 2.4 illustrates this process for the construction of a tree with depth limit 2. Here the first argument of the + root node happens to be a terminal. This closes off that branch preventing it from growing any more before it reached the depth limit.

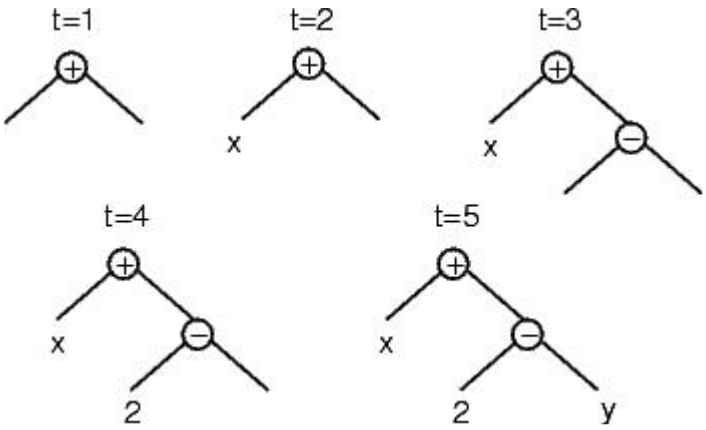


Figure 4.3 Creation of a five node tree using the grow initialisation method with a maximum depth of 2 (t = time). A terminal is chosen at t = 2, causing the left branch of the root to be closed at that point even though the maximum depth had not been reached.

---

procedure: gen_rnd_expr(func_set,term_set,max_d,method)

1: if max_d = 0 or $\left(\text{method = grow and rand() < } \frac{|\text{termxset}|}{|\text{termxset}|+|\text{funcxset}|}\right)$ then

2: expr = choose_random_element(term_set)

3: else

4: func = choose_random_element(func_set)

5: for i = 1 to arity(func) do

6: arg_i = gen_rnd_expr(func_set, term_set, max_d - 1, method);

7: end for

8: expr = (func, arg_1, arg_2, ...);

9: end if

10: return expr

Notes: func_set is a function set, term_set is a terminal set, max_d is the maximum allowed depth for expressions, method is either full or grow, expr is the generated expression in prefix notation and rand() is a function that returns random numbers uniformly distributed between 0 and 1.

Because neither the grow or full method provide a very wide array of sizes or shapes on their own, Koza (1992) proposed a combination called ramped half-and-half. Half the initial population is constructed using full and half is constructed using grow. This is done using a range of depth limits (hence the term "ramped") to help ensure that we generate trees having a variety of sizes and shapes.

While these methods are easy to implement and use, they often make it difficult to control the statistical distributions of important properties such as the sizes and shapes of the generated trees. For example, the sizes and shapes of the trees generated via the grow method are highly sensitive to the sizes of the function and terminal sets. If, for example, one has significantly more terminals than functions, the grow method will almost always generate very short trees regardless of the depth limit. Similarly, if the number of functions is considerably greater than the number of terminals, then the grow method will behave quite similarly to the full method. The arities of the functions in the primitive set also influence the size and shape of the trees produced by grow. The initial population need not be entirely random. If something is known about likely properties of the desired solution, trees having these properties can be used to seed the initial population.

4.1.3 Selection

As with most evolutionary algorithms, genetic operators in genetic programs are applied to individuals that are probabilistically selected based on fitness. That is, better individuals are more likely to have more child programs than inferior

individuals. The most commonly employed method for selecting individuals in genetic programming is tournament selection, which is discussed below, followed by fitness-proportionate selection, but any standard evolutionary algorithm selection mechanism can be used.

In tournament selection a number of individuals are chosen at random from the population. These are compared with each other and the best of them is chosen to be the parent. When doing crossover, two parents are needed and, so, two selection tournaments are made. Note that tournament selection only looks at which program is better than another. It does not need to know how much better. This effectively automatically rescales fitness, so that the selection pressure on the population remains constant. Thus, a single extraordinarily good program cannot immediately swamp the next generation with its children; if it did, this would lead to a rapid loss of diversity with potentially disastrous consequences for a run. Conversely, tournament selection amplifies small differences in fitness to prefer the better program even if it is only marginally superior to the other individuals in a tournament.

An element of noise is inherent in tournament selection due to the random selection of candidates for tournaments. So, while preferring the best, tournament selection does ensure that even average-quality programs have some chance of having children. Since tournament selection is easy to implement and provides automatic fitness rescaling, it is commonly used in genetic programming

Considering that selection has been described many times in the evolutionary algorithms literature, we will not provide details of the numerous other mechanisms that have been proposed. Goldberg, 1989, for example, describes fitness-proportionate selection, stochastic universal sampling and several others.

4.1.4 Recombination and Mutation

Genetic programming departs significantly from other evolutionary algorithms in the implementation of the operators of crossover and mutation. The most commonly used form of crossover is subtree crossover. Given two parents, subtree crossover

randomly (and independently) selects a crossover point (a node) in each parent tree. Then, it creates the offspring by replacing the subtree rooted at the crossover point in a copy of the first parent with a copy of the subtree rooted at the crossover point in the second parent. Copies are used to avoid disrupting the original individuals. This way, if selected multiple times, they can take part in the creation of multiple offspring programs. Note that it is also possible to define a version of crossover that returns two offspring, but this is not commonly used.

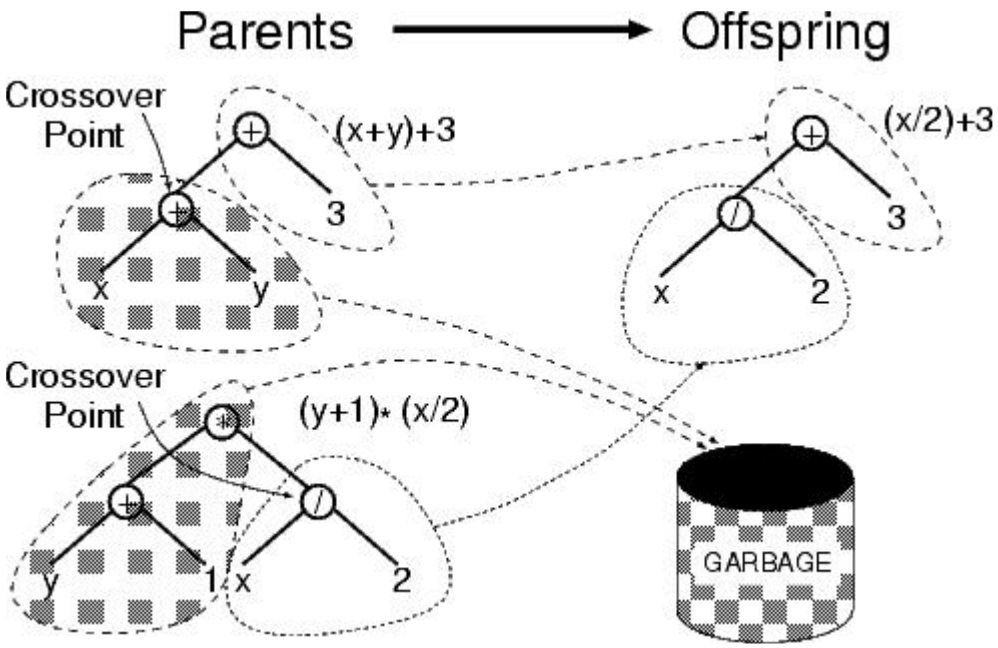


Figure 4.4 Example of subtree crossover. Note that the trees on the left are actually copies of the parents. So, their genetic material can freely be used without altering the original individuals.

Often crossover points are not selected with uniform probability. Typical genetic program primitive sets lead to trees with an average branching factor (the number of children of each node) of at least two, so the majority of the nodes will be leaves. Consequently the uniform selection of crossover points leads to crossover operations frequently exchanging only very small amounts of genetic material (i.e., small subtrees); many crossovers may in fact reduce to simply swapping two leaves. To counter this, suggested the widely used approach of choosing functions 90% of the

time and leaves 10% of the time. Many other types of crossover and mutation of genetic program trees are possible.

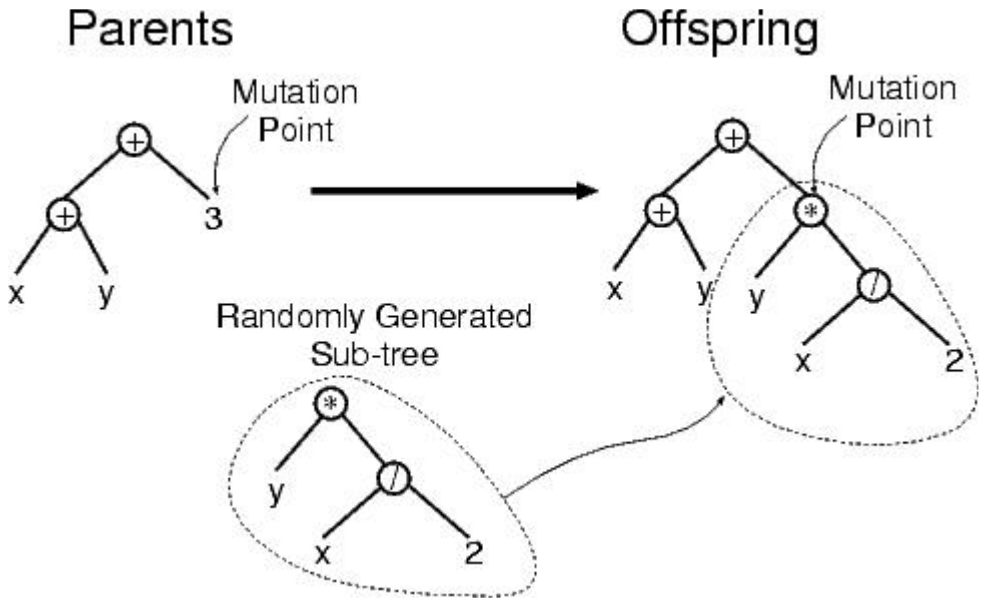


Figure 4.5 Example of subtree mutation.

The most commonly used form of mutation in genetic programming (which we will call subtree mutation) randomly selects a mutation point in a tree and substitutes the subtree rooted there with a randomly generated subtree. Subtree mutation is sometimes implemented as crossover between a program and a newly generated random program; this operation is also known as "headless chicken" crossover.

Another common form of mutation is point mutation, which is genetic programming's rough equivalent of the bit-flip mutation used in genetic algorithms. In point mutation, a random node is selected and the primitive stored there is replaced with a different random primitive of the same arity taken from the primitive set. If no other primitives with that arity exist, nothing happens to that node (but other nodes may still be mutated). When subtree mutation is applied, this involves the modification of exactly one subtree. Point mutation, on the other hand, is typically applied on a per-node basis. That is, each node is considered in turn and,

with a certain probability, it is altered as explained above. This allows multiple nodes to be mutated independently in one application of point mutation.

The choice of which of the operators described above should be used to create an offspring is probabilistic. Operators in genetic programming are normally mutually exclusive (unlike other evolutionary algorithms where offspring are sometimes obtained via a composition of operators). Their probability of application is called operator rates. Typically, crossover is applied with the highest probability, the crossover rate often being 90% or higher. On the contrary, the mutation rate is much smaller, typically being in the region of 1%.

When the rates of crossover and mutation add up to a value p which is less than 100%, an operator called reproduction is also used, with a rate of 1 - p. Reproduction simply involves the selection of an individual based on fitness and the insertion of a copy of it in the next generation.

4.1.5 Terminal Set

While it is common to describe genetic programming as evolving programs, genetic programming is not typically used to evolve programs in the familiar Turing-complete languages humans normally use for software development. It is instead more common to evolve programs (or expressions or formulae) in a more constrained and often domain-specific language. The first two preparatory steps, the definition of the terminal and function sets, specify such a language. That is, together they define the ingredients that are available to a genetic program to create computer programs.

The terminal set may consist of:

- the program's external inputs. These typically take the form of named variables (e.g., x, y).
- functions with no arguments. These may be included because they return different values each time they are used, such as the function rand() which returns random numbers, or a function dist_to_wall() that returns the distance

to an obstacle from a robot that a genetic programming system is controlling. Another possible reason is because the function produces side effects. Functions with side effects do more than just return a value: they may change some global data structures, print or draw something on the screen, control the motors of a robot, etc.

- constants. These can be pre-specified, randomly generated as part of the tree creation process, or created by mutation.

Using a primitive such as rand can cause the behaviour of an individual program to vary every time it is called, even if it is given the same inputs. This is desirable in some applications. However, we more often want a set of fixed random constants that are generated as part of the process of initialising the population. This is typically accomplished by introducing a terminal that represents an ephemeral random constant. Every time this terminal is chosen in the construction of an initial tree (or a new subtree to use in an operation like mutation), a different random value is generated which is then used for that particular terminal, and which will remain fixed for the rest of the run.

4.1.6 Function Set

The function set used in genetic programming is typically driven by the nature of the problem domain. In a simple numeric problem, for example, the function set may consist of merely the arithmetic functions (+, -, *, /). However, all sorts of other functions and constructs typically encountered in computer programs can be used. Table 3.1 shows a sample of some of the functions one sees in the genetic programming literature. Sometimes the primitive set includes specialised functions and terminals which are designed to solve problems in a specific problem domain. For example, if the goal is to program a robot to mop the floor, then the function set might include such actions as move, turn, and swish-the-mop.

Table 3.1: Examples of primitives in genetic program function and terminal sets.

Function Set

Kind of Primitive	Example(s)
Arithmetic	+, *, /
Mathematical	sin, cos, exp
Boolean	AND, OR, NOT
Conditional	IF-THEN-ELSE
Looping	FOR, REPEAT
⋮	⋮

Terminal Set

Kind of Primitive	Example(s)
Variables	x, y
Constant values	3, 0.45
0-arity functions	rand, go_left

For genetic programming to work effectively, most function sets are required to have an important property known as closure (Koza, 1992), which can in turn be broken down into the properties of type consistency and evaluation safety.

Type consistency is required because subtree crossover) can mix and join nodes arbitrarily. As a result, it is necessary that any subtree can be used in any of the argument positions for every function in the function set, because it is always possible that subtree crossover will generate that combination. It is thus common to require that all the functions be type consistent, i.e., they all return values of the same type, and that each of their arguments also have this type. For example +, -, *, and / can be defined so that they each take two integer arguments and return an integer. Sometimes type consistency can be weakened somewhat by providing an automatic conversion mechanism between types. We can, for example, convert numbers to Booleans by treating all negative values as false, and non-negative values as true. However, conversion mechanisms can introduce unexpected biases into the search process, so they should be used with care.

The type consistency requirement can seem quite limiting but often simple restructuring of the functions can resolve apparent problems. For example, an if function is often defined as taking three arguments: the test, the value to return if the test evaluates to true and the value to return if the test evaluates to false. The first of these three arguments is clearly Boolean, which would suggest that if can't be used with numeric functions like +. This, however, can easily be worked around by providing a mechanism to convert a numeric value into a Boolean automatically as discussed above. Alternatively, one can replace the 3-input if with a function of four (numeric) arguments a,b,c,d. The 4-input if implements "If a < b then return value c otherwise return value d".

An alternative to requiring type consistency is to extend the genetic programming system. Crossover and mutation might explicitly make use of type information so that the children they produce do not contain illegal type mismatches. When mutating a legal program, for example, mutation might be required to generate a subtree which returns the same type as the subtree it has just deleted.

The other component of closure is evaluation safety. Evaluation safety is required because many commonly used functions can fail at run time. An evolved expression might, for example, divide by 0, or call MOVE_FORWARD when facing a wall or precipice. This is typically dealt with by modifying the normal behaviour of primitives. It is common to use protected versions of numeric functions that can otherwise throw exceptions, such as division, logarithm, exponential and square root. The protected version of a function first tests for potential problems with its input(s) before executing the corresponding instruction; if a problem is spotted then some default value is returned. Protected division (often notated with %) checks to see if its second argument is 0. If so, % typically returns the value 1 (regardless of the value of the first argument). Similarly, in a robotic application a MOVE_AHEAD instruction can be modified to do nothing if a forward move is illegal or if moving the robot might damage it.

An alternative to protected functions is to trap run-time exceptions and strongly reduce the fitness of programs that generate such errors. However, if the likelihood of generating invalid expressions is very high, this can lead to too many individuals in the population having nearly the same (very poor) fitness. This makes it hard for selection to choose which individuals might make good parents.

One type of run-time error that is more difficult to check for is numeric overflow. If the underlying implementation system throws some sort of exception, then this can be handled either by protection or by penalising as discussed above. However, it is common for implementation languages to ignore integer overflow quietly and simply wrap around. If this is unacceptable, then the genetic program implementation must include appropriate checks to catch and handle such overflows.

There is one more property that primitives sets should have: sufficiency. Sufficiency means it is possible to express a solution to the problem at hand using the elements of the primitive set. Unfortunately, sufficiency can be guaranteed only for those problems where theory, or experience with other methods, tells us that a solution can be obtained by combining the elements of the primitive set.

As an example of a sufficient primitive set consider {AND, OR, NOT, x1, x2, ..., xN}. It is always sufficient for Boolean induction problems, since it can produce all

Boolean functions of the variables x1, x2, ..., xN. An example of insufficient set is {+, -, *, /, x, 0, 1, 2}, which is unable to represent transcendental functions. The function exp(x), for example, is transcendental and therefore cannot be expressed as a rational function (basically, a ratio of polynomials), and so cannot be represented exactly by any combination of {+, -, *, /, x, 0, 1, 2}. When a primitive set is insufficient, the genetic program can only develop programs that approximate the desired one. However, in many cases such an approximation can be very close and good enough for the user's purpose. Adding a few unnecessary primitives in an attempt to ensure sufficiency does not tend to slow down a genetic program that much, although there are cases where it can bias the system in unexpected ways.

There are many problems where solutions cannot be directly cast as computer programs. For example, in many design problems the solution is an artefact of some type: a bridge, a circuit, an antenna, a lens, etc. Genetic programming has been applied to problems of this kind by using a trick: the primitive set is set up so that the evolved programs construct solutions to the problem. This is analogous to the process by which an egg grows into a chicken. For example, if the goal is the automatic creation of an electronic controller for a plant, the function set might include common components such as integrator, differentiator, lead, lag, and gain, and the terminal set might contain reference, signal, and plant output. Each of these primitives, when executed, inserts the corresponding device into the controller being built. If, on the other hand, the goal is to synthesise analogue electrical circuits, the function set might include components such as transistors, capacitors, resistors, etc.

4.1.7 Fitness Function

The first two preparatory steps define the primitive set for a genetic program, and therefore indirectly define the search space the genetic program will explore. This includes all the programs that can be constructed by composing the primitives in all possible ways. However, at this stage, we still do not know which elements or regions of this search space are good. I.e., which regions of the search space include programs that solve, or approximately solve the problem. This is the task of the fitness measure, which is our primary (and often sole) mechanism for giving a high-

level statement of the problem's requirements to the genetic programming system. For example, suppose the goal is to get the genetic program to synthesise an amplifier automatically. Then the fitness function is the mechanism which tells the genetic program to synthesise a circuit that amplifies an incoming signal. (As opposed to evolving a circuit that suppresses the low frequencies of an incoming signal, or computes its square root, etc. etc.)

Fitness can be measured in many ways. For example, in terms of: the amount of error between its output and the desired output; the amount of time (fuel, money, etc.) required to bring a system to a desired target state; the accuracy of the program in recognising patterns or classifying objects; the payoff that a game-playing program produces; the compliance of a structure with user-specified design criteria.

There is something unusual about the fitness functions used in genetic programming that differentiates them from those used in most other evolutionary algorithms. Because the structures being evolved in genetic programming are computer programs, fitness evaluation normally requires executing all the programs in the population, typically multiple times. While one can compile the genetic program programs that make up the population, the overhead of building a compiler is usually substantial, so it is much more common to use an interpreter to evaluate the evolved programs.

Interpreting a program tree means executing the nodes in the tree in an order that guarantees that nodes are not executed before the value of their arguments (if any) is known. This is usually done by traversing the tree recursively starting from the root node, and postponing the evaluation of each node until the values of its children (arguments) are known. Other orders, such as going from the leaves to the root, are possible. If none of the primitives have side effects, the two orders are equivalent.3 This depth-first recursive process is illustrated in Figure 3.1 . Algorithm 3.1 gives a pseudocode implementation of the interpretation procedure. The code assumes that programs are represented as prefix-notation expressions and that such expressions can be treated as lists of components.

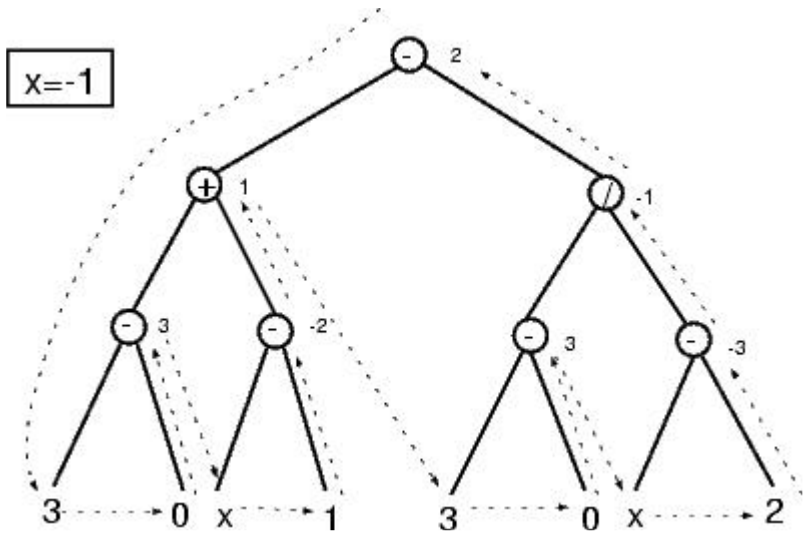


Figure 4.6 Example interpretation of a syntax tree (the terminal x is a variable and has a value of -1). The number to the right of each internal node represents the result of evaluating the subtree root at that node.

procedure: eval(expr)

1: if expr is a list then

2: proc = expr(1) {Non-terminal: extract root}

3: if proc is a function then

4: value = proc(eval(expr(2)), eval(expr(3)), ...) {Function: evaluate arguments}

5: else

6: value = proc(expr(2), expr(3), ...) {Macro: don't evaluate arguments}

7: end if

8: else

9: if expr is a variable or expr is a constant then

10: value = expr {Terminal variable or constant: just read the value}

11: else

12: value = expr() {Terminal 0-arity function: execute}

13: end if

14: end if

15: return value

Notes: expr is an expression in prefix notation, expr(1) represents the primitive at the root of the expression, expr(2) represents the first argument of that primitive, expr(3) represents the second argument, etc.

Algorithm 3.1: Interpreter for genetic programming

In some problems we are interested in the output produced by a program, namely the value returned when we evaluate the tree starting at the root node. In other problems we are interested in the actions performed by a program composed of functions with side effects. In either case the fitness of a program typically depends on the results produced by its execution on many different inputs or under a variety of different conditions. For example the program might be tested on all possible combinations of inputs x1, x2, ..., xN. Alternatively, a robot control program might be tested with the robot in a number of starting locations. These different test cases typically contribute to the fitness value of a program incrementally, and for this reason are called fitness cases.

Another common feature of genetic programming fitness measures is that, for many practical problems, they are multi-objective, i.e., they combine two or more different elements that are often in competition with one another. The area of multi-objective optimisation is a complex and active area of research in genetic programming and machine learning in general.

4.1.8 Parameters

The fourth preparatory step specifies the control parameters for the run. The most important control parameter is the population size. Other control parameters include the probabilities of performing the genetic operations, the maximum size for programs and other details of the run.

It is impossible to make general recommendations for setting optimal parameter values, as these depend too much on the details of the application. However, genetic programming is in practice robust, and it is likely that many different parameter values will work. As a consequence, one need not typically spend a long time tuning the genetic programming for it to work adequately.

It is common to create the initial population randomly using ramped half-and-half with a depth range of 2-6. The initial tree sizes will depend upon the number of the functions, the number of terminals and the arities of the functions. However, evolution will quickly move the population away from its initial distribution.

Traditionally, 90% of children are created by subtree crossover. However, the use of a 50-50 mixture of crossover and a variety of mutations also appears to work well.

In many cases, the main limitation on the population size is the time taken to evaluate the fitnesses, not the space required to store the individuals. As a rule one prefers to have the largest population size that your system can handle gracefully; normally, the population size should be at least 500, and people often use much larger populations. Often, to a first approximation, a genetic program runtime can be estimated by the product of: the number of runs R, the number of generations G, the size of the population P, the average size of the programs s and the number of fitness cases F.

Typically, the number of generations is limited to between ten and fifty; the most productive search is usually performed in those early generations, and if a solution hasn't been found then, it's unlikely to be found in a reasonable amount of time. The folk wisdom on population size is to make it as large as possible, but there are those

who suggest using many runs with much smaller populations instead. Some implementations do not require arbitrary limits of tree size. Even so, because of bloat, it is common to impose either a size or a depth limit or both/

Sometimes the number of fitness cases is limited by the amount of training data available. In this case, the fitness function should use all of it. Then the fitness function may be reduced to use just a subset of the training data. This does not necessarily have to be done manually as there are a number of algorithms that dynamically change the test set as the genetic program runs.

4.2 Background subtraction

Background subtraction in image processing is a method to separate stationary background objects from moving foreground objects. At its most basic form, it works by subtracting a new frame from a calculated reference image and thresholding the result [63]–[66]. The background can be estimated accurately using an adaptive Gaussian mixture model [63], [64], [66]. A Gaussian mixture model is a probability density function represented as a liner combination of Gaussian component densities. The main advantage of using an adaptive Gaussian mixture model to estimate the background is that it accounts for slow changes in illumination and shadows [64]. With a strong estimate of the background of a scene, excluding foreground moving objects, the scene illuminant can be estimated with greater accuracy, eliminating noise or clutter that would otherwise distort results [67].

4.3 Genetic programming applied to the colour constancy problem

A Genetic program system has been designed to find solutions for colour constancy on a set of stationary surveillance images. The external input for the genetic programs in this case is an image. All functions operating on the input variable must therefore return the same type (image / matrix) [55]. Each genetic program contains at least one genetic tree. Each genetic tree describes a function of each channel (e.g. a program operating on an RGB image would have 3 genetic trees, 1 for each channel).

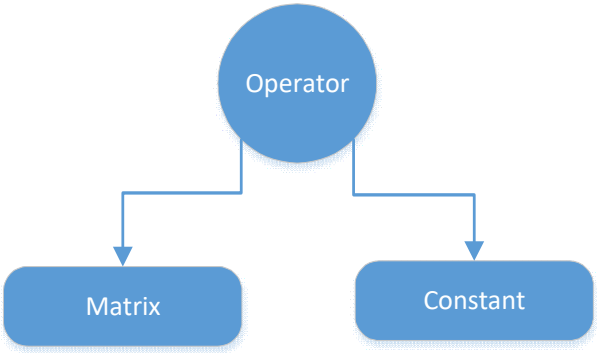
$$corrected \begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} f(R) \\ f(G) \\ f(B) \end{bmatrix}$$

The functions of each channel can be expressed as:

$$f_i = \Delta c_i$$

Where Δc represents the change between a channel in the original image and the corrected image, i is the channel number.

Each genetic tree is made up of genetic tree items (nodes). Each item has a type and references to its child items [55]. A genetic item type can be a matrix (variable), constant or operator. If the item is a constant or matrix it can be represented as a leaf node of its parent, having no children. If the item is an operator it must have two children in order to perform the operation. Below is the most basic tree structure that can be formed.



The genetic programs take a frame from a camera as input and perform random operations on each channel. The median angular error between the output image and the target image is used as the fitness case. The programs with the least angular error (fittest) are then selected for crossover breeding and the process is repeated over as many generations as needed to get to a suitable solution.

The angular error between pixels of two RGB images is calculated as follows:

$$a = \sqrt{R1^2 + G1^2 + B1^2}$$

$$b = \sqrt{R2^2 + G2^2 + B2^2}$$

$$e = \sqrt{(R1 - R2)^2 + (G1 - G2)^2 + (B1 - B2)^2}$$

$$\theta_i = cos^{-1} \frac{a_i^2 + b_i^2 - e_i^2}{2a_i b_i}$$

Therefore the fitness of a program can be expressed as:

$$fi = median(cos^{-1} \frac{a_i^2 + b_i^2 - e_i^2}{2a_i b_i})$$

Where fi is an individual program, a is the corrected pixel vector, b is the target pixel vector and e is the error vector.

The programs size (number of nodes) is restricted using a parsimony value which acts as a deterrent to increasing the program size during the selection process [55]. This is used to decrease overall computational cost during the evaluation of a program i.e. a simple program will take less time to compute. The fitness of a program with parsimony pressure can therefore be expressed as:

$$fp = fi - (c \times li)$$

Where fp is the fitness of a single program with parsimony; c is the parsimony constant and li is the individual size of a program.

The programs fitness is tested against other programs using tournament selection. A tournament size of 2 participants (best of 2 randomly selected programs is allowed to breed) is used to ensure there is no bias towards a local optimum solution. Mutation is used to allow the program to explore other solutions in the search space and not focus on a local solution. The mutation rate is the chance of a node changing

randomly during the breeding process. A crossover rate of 0.5 is used to ensure a fair selection of each programs data during the breeding process.

At the top level, the system uses an adaptive Gaussian mixture model to segment the background of the scene. In theory the background of a stationary scene remains constant; however, due to changing lighting conditions, surfaces in the background could appear to be of a different colour entirely [63]–[65]. The adaptive Gaussian mixture model described by Zivkovich et al also computes a mask for shadow detection [63]. Removal of shadows is extremely beneficial to environmentally robust camera systems as it can obtain a better understanding of why an object has perceived colour change [68]–[70]. The system takes snapshots of the background model at different times of the day. Each snapshot is then used by the genetic programs, finding functions to correct each channel of the image to get from snapshot A to snapshot B.

The best solution produced by the genetic programs can then be applied to the foreground of the scene. As previously mentioned, for an RGB image a genetic program would comprise of three genetic trees, each a function of a channel of the image. Each channel in the foreground image uses the same functions computed from the background model. This means that foreground objects colours at one time of day can be adjusted to match those of another time of day, making a scene invariant to lighting conditions.

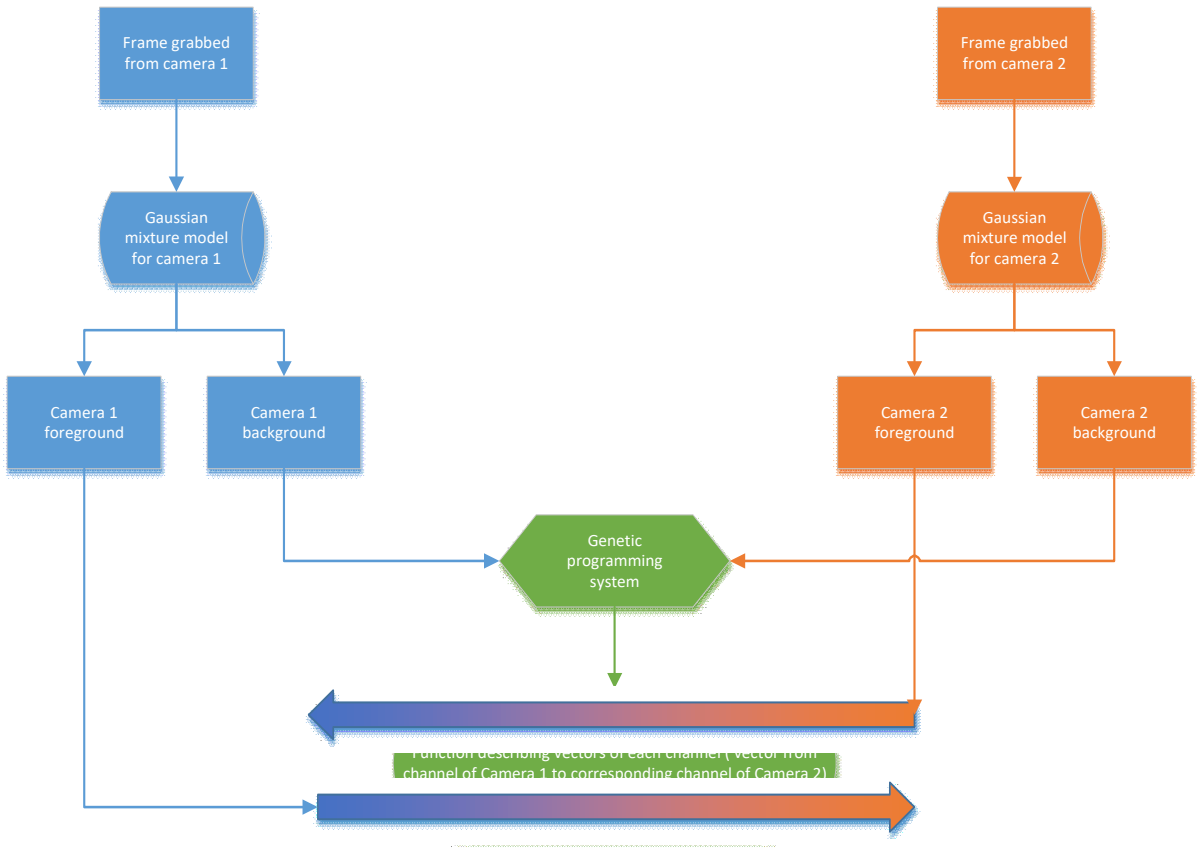


Figure 4.7 Multi-camera genetic program system concept

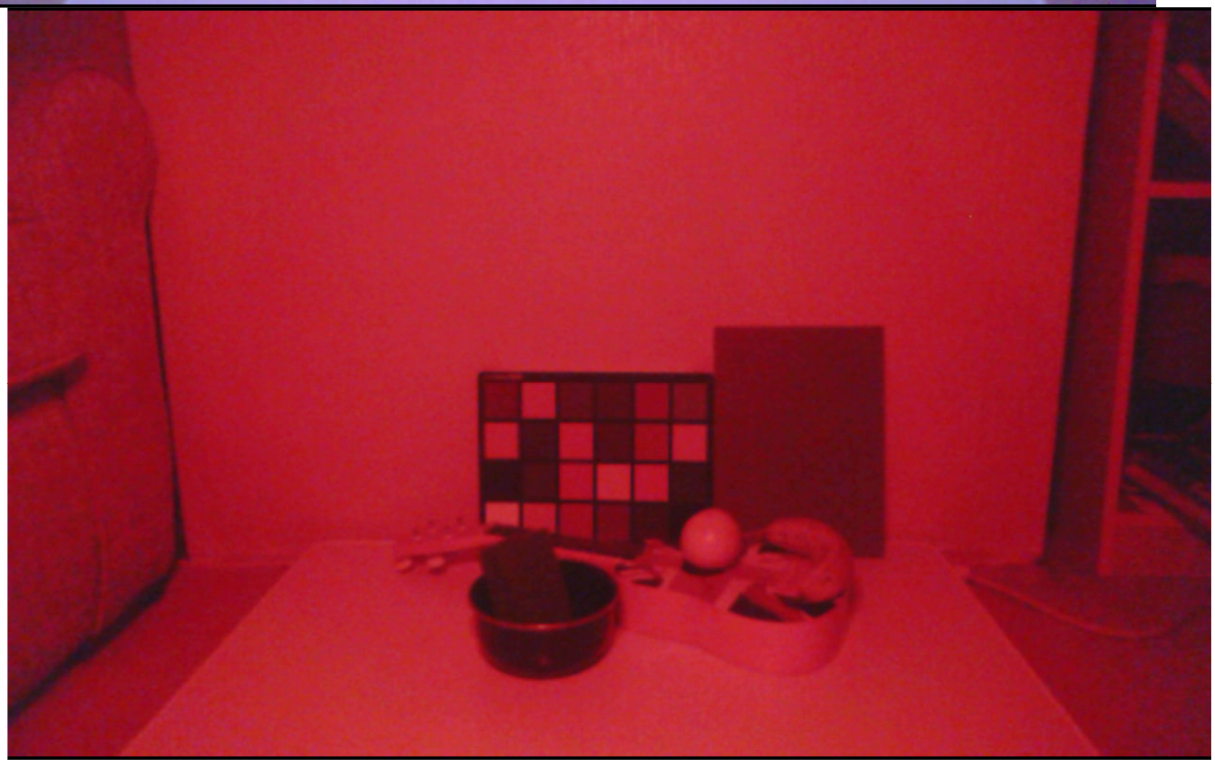
Traditional colour constancy algorithms, such as Grey-world and Grey-edge, attempt to adjust an images illuminant to be achromatic [38], [41], [44]. However, if you consider a light source with a limited spectrum, areas in the image that appear black (because no light is being reflected) become impossible to reconstruct in terms of colour [71]. For example, if a blue light was used to illuminate a scene, objects that normally appear red, yellow or green would appear black. It would be impossible to say, without prior knowledge, that an object that appears black in the blue illuminant scene has a specific colour in a standard D60 illuminant (day light) scene because it could be any of the above mentioned colours, as well as the possibility of being black.

The correct approach would therefore be to adjust the scene that has a wider range of colours (more information) to that of the scene with less colour information. A similar approach can be found in the process of 2D gamut mapping, where Forsyth suggests a scene with the largest gamut (most amount of colours) should be used

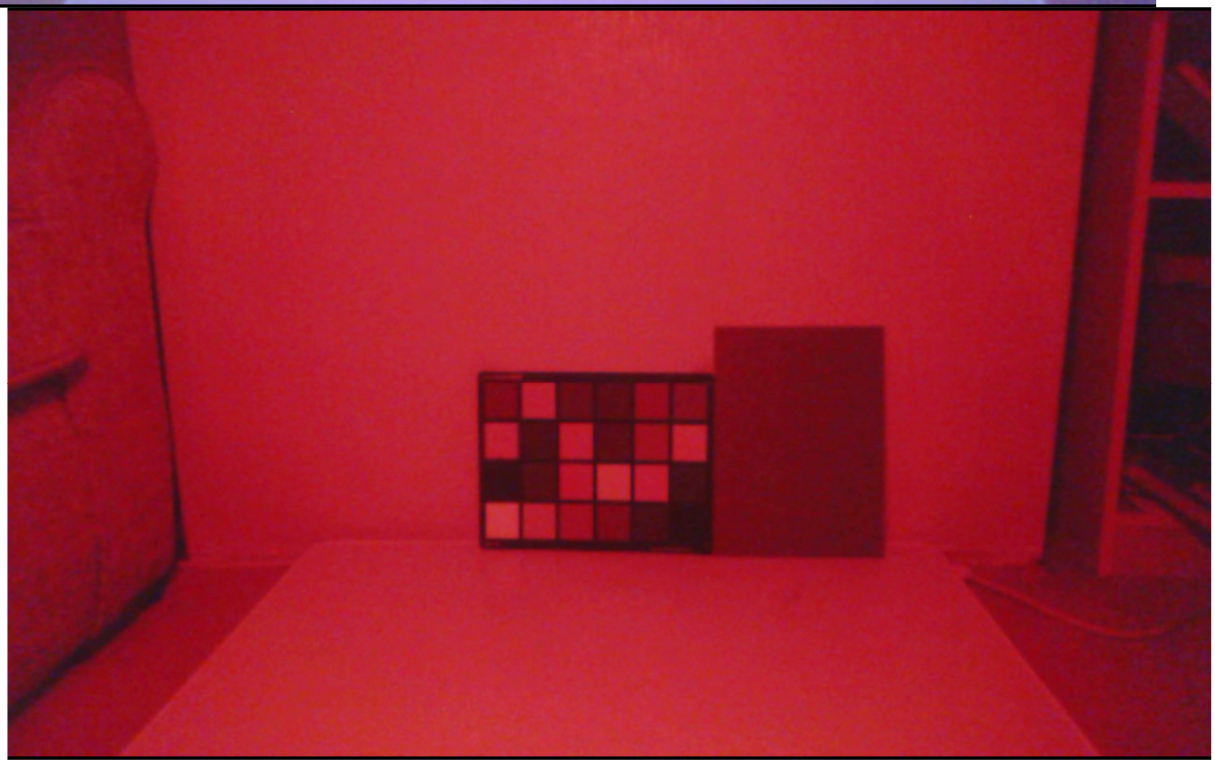
preferably than a scene with a smaller gamut [41], [43], [45], [72]. Going back to the blue illuminant scenario, the other scene would be adjusted to try and match the lighting in the blue scene.

4.4 Experiment

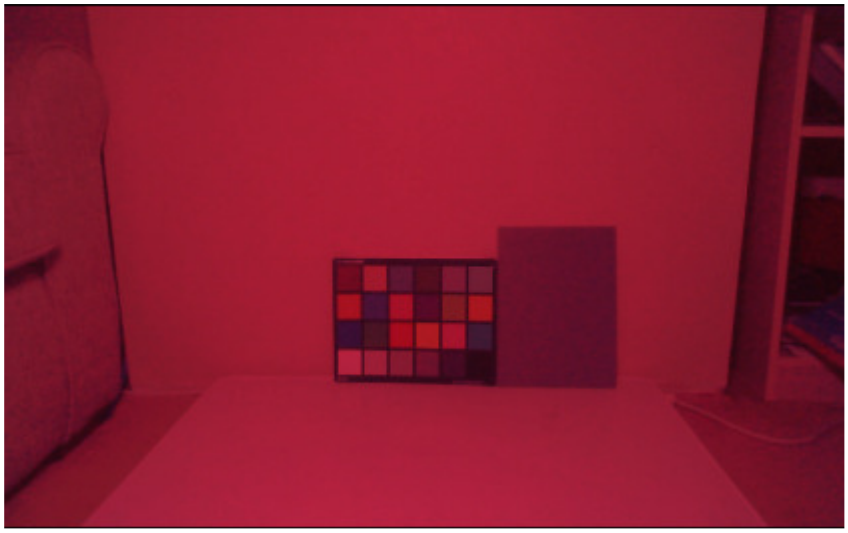
An experiment was carried out to test the practicality of the system in a real-life scenario. The experiment is designed to measure the improvements this colour constancy system has on feature matching tracking algorithms such as SURF and SIFT. In order to speed up the process of evaluating each genetic program, each program is assigned to a thread so that multiple programs can be evaluated in parallel [73]. An incandescent illuminated scene and a low temperature LED lit scene with objects are used to test how well the tracking algorithms perform in matching the objects before and after colour correction from the genetic programs solution. The maximum initial depth of the trees to be generated is set to 8. The larger the population the more likely a good solution is found, however, due to memory and computational power, a population of 1000 was found to work sufficiently.



The genetic algorithm takes the Gaussian background model of the incandescent illuminated scene as input and uses the Gaussian background model of the low temperature LED lit scene as the target.



The best genetic programs solution will try to match the target as much as possible creating a similar image.

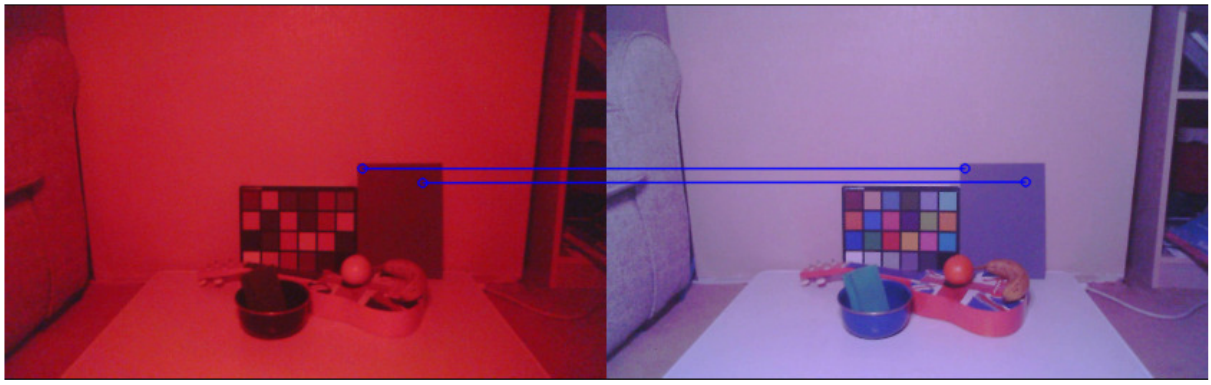


The programs solution is then applied to the input image containing objects, transforming the objects colours in the input image to match those of the objects in the LED lit scene.

This estimate of what the objects would appear to look like under the LED illuminant can then be used for feature matching.

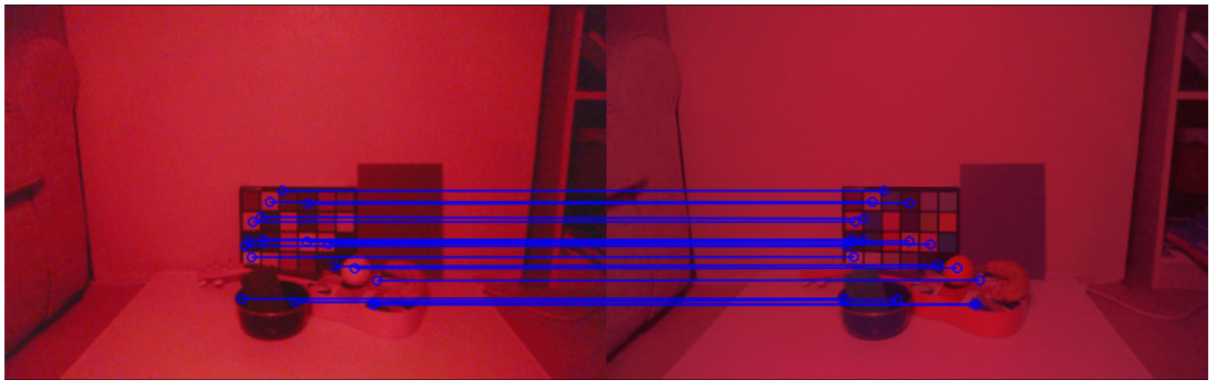
A SURF feature detector was used to extract keypoint descriptors in both scenes before and after processing the image colours. A FLANN based matcher was used to

match the keypoint descriptors in both scenes. The matches with Hessian distance less than double the minimum distance are used as “good matches” [27].



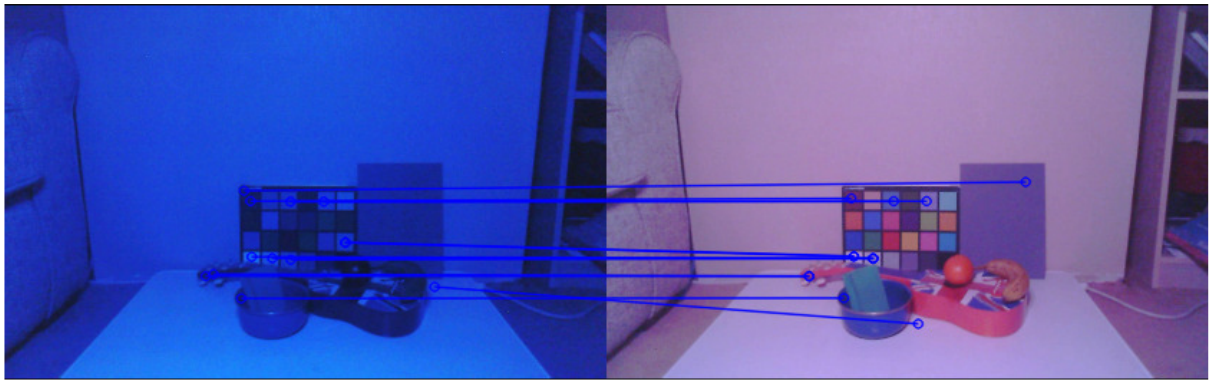
This image shows that only two good matches were found without colour correction.

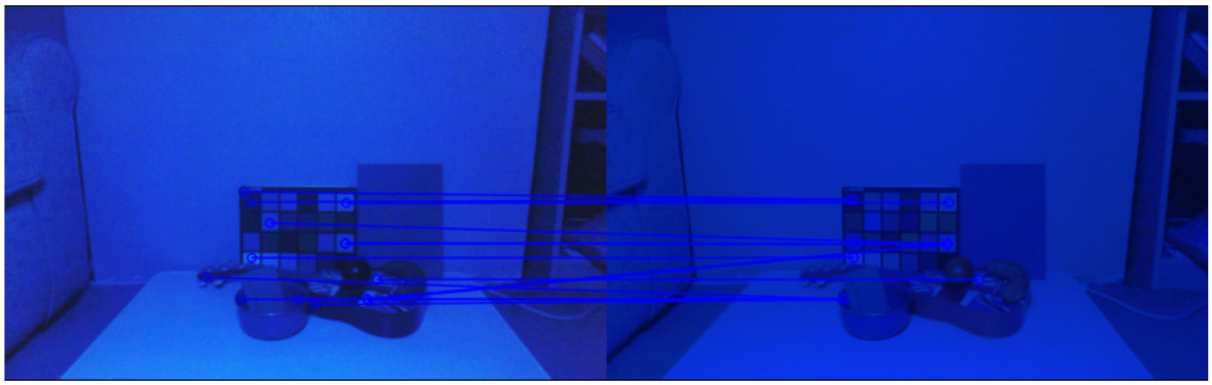
With colour correction however, all the objects in the scene are detected and matched correctly.



The median angular error between the output and target images was calculated to be 4.8307 degrees.

Below is another example using a high temperature blue LED illuminant.



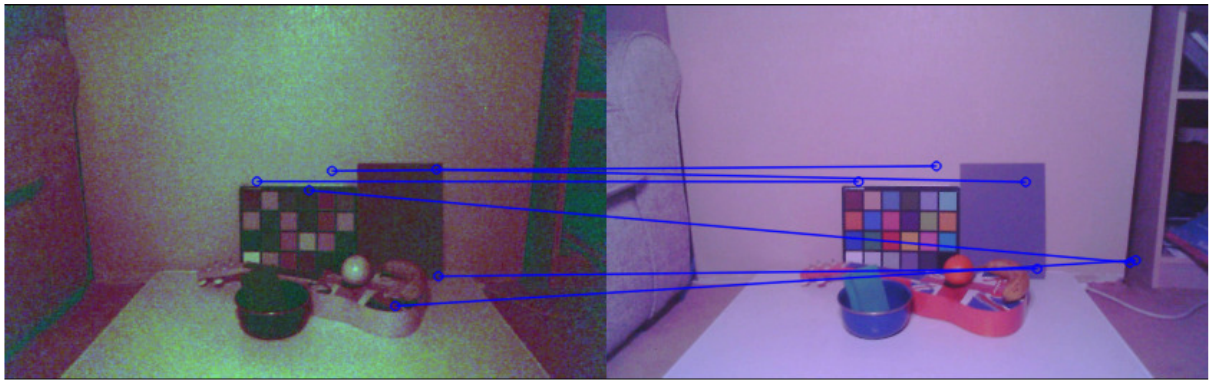


Although this result shows little difference in terms of key point detection, the number of correct matches in the colour corrected image is much greater.

To test how well this system performs compared to conventional colour constancy algorithms, the same experiment was repeated only using outputs from the Grey World, Max-RGB, Shades of Grey and Grey Edge algorithms.

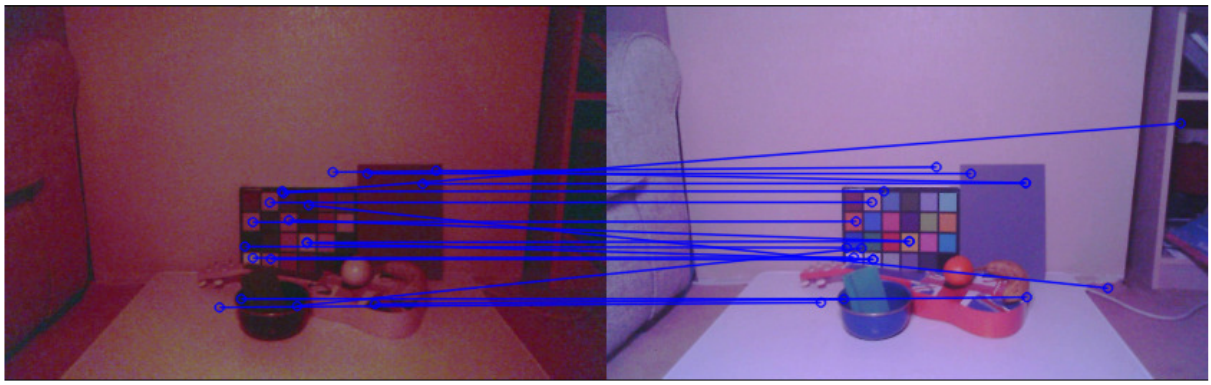
Using the red LED illuminant scene as input the conventional algorithms produced the following outputs.

Grey-World output



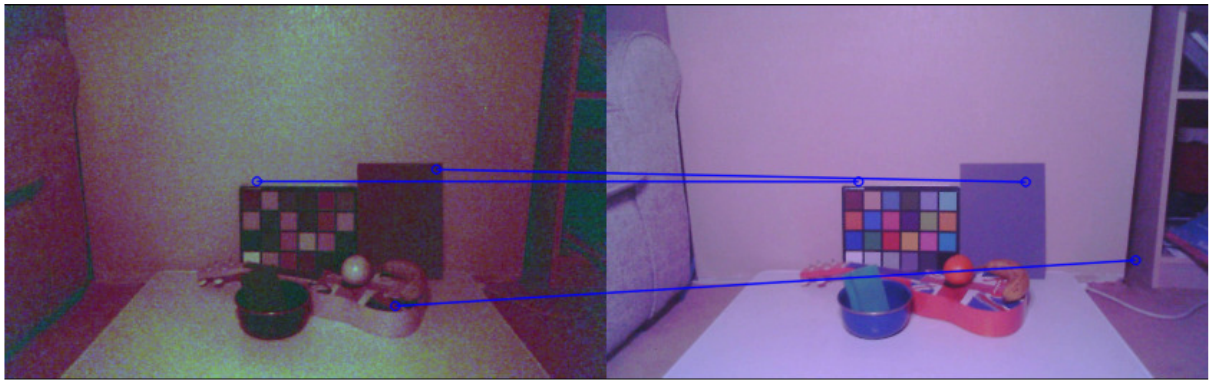
The median angular error between the Grey-World output and the target image is calculated as 11.3341 degrees. Visually, there also appears to be an increased contrast in the Grey-World output, amplifying noise intensity, “creating” false features.

Max-RGB output



The median angular error between the Max-RGB output and the target image is 18.5686 degrees. Although there are a large number of correct matches, there also appears to be a large number of false matches.

Shades of Grey output



The median angular error between the shades of grey output and the target image is 10.5546 degrees. This output is visually similar to that of the grey-world output, again, creating false features resulting in false matches.

Grey-Edge output

The median angular error between the grey-edge output and the target image is calculated as 9.9196 degrees. Only one good match is found in this instance.

4.5 Conclusions

From experiments it is clear that Genetic Programming can be used to find colour constancy algorithms for images with different illuminants, outperforming traditional colour constancy techniques. The main advantage of this technique is that it can be used on images with extremely low or high temperature illumination.

However, its main disadvantage is that without a scene with a large spectrum of colour present, it fails to estimate colours it has no reference to. Also, it cannot be used the same way as traditional colour constancy algorithms, attempting to adjust the illuminant to be achromatic. It needs two images in order to calculate the functions describing the changes in each channel.

The genetic program system that has been implemented exploits the use of multi-threading made available to processors with multiple cores. This multi-threading approach, as expected, significantly improves processing time. This improvement in processing time means that the usually time consuming testing of the genetic programs can be done much faster providing accurate solutions at similar speeds to low computational colour constancy algorithms.

Chapter 5 : CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

The objective of this research is to create a highly robust multiple camera tracking system with invariance to changes in lighting, scale and orientation of the target being tracked. This research also explores methods for artificial learning by incorporating genetic programming into pre-processing colour correction steps. C++, a cross-platform programming language. Is used for the development of the systems in Chapters 3 and 4 of this thesis. The design focus of the software is robustness and re-usability of individual classes with the intent of further improvements in the future.

The tracking technology chosen in this thesis is feature detection. Chapter 1 describes current feature detection methods and possible ways to improve tracking accuracy. In Chapter 2, a novel colour constancy transformation algorithm is developed to improve feature detection during lighting changes, shadow edges or camera colour response differences. The algorithm uses pre-existing colour constancy estimators defined in Chapter 1 as well as a median-based estimate to correct the colour in an image. A rotational transform is applied to the colours in Opponent colour space instead of using a traditional diagonal transform. From the

experiments it is clear that the devised transform maintains colour edge contrast better than a diagonal transform. Maintaining such colour contrast is extremely beneficial when using feature detection algorithms as true features could be lost, or false features created, hindering the accuracy of a tracking system.

To address the problem of colour constancy across multiple cameras, in Chapter 3, an artificially intelligent system is developed using genetic programming to automatically find a solution to achieving colour constancy between two images. Apart from providing accurate solutions, the genetic programming system has a relatively quick execution time due to exploiting multi-threading and parallel computing. The software modules / classes have been designed to be reusable for adding new functionality in the future.

Chapter 4 brings all of the pre-processing steps and tracking techniques together into one system to ensure an environmentally robust multiple camera tracking system. The main pre-processing emphasis being to achieve colour constancy, as highly robust features can be used for tracking. The combination of achieving colour constancy (or as close to as can be) and gathering scale and orientation invariant features in a shade / shadow invariant colour space results in a tracking system that is robust to scale, orientation, occlusion, lighting, shade and shadow fluctuations, thus meeting the requirements of the objectives and research outputs described in chapter 1.

5.2 Future Work

When it comes to enhancing the work that has been done in this thesis, there are 2 prospective areas that would benefit from further work.

The first aspect is the improvement of the performance of the genetic programming system. With limited functional sets, only certain solutions can be explored. Adding more advanced functions such as sine, cosine, square root etc would allow the genetic program to explore a wider search space, providing possibly better solutions that might not have been thought of. A new method of estimating lighting could be devised by changing the fitness algorithm so that it benefits genetic programs whose solution solves the case.

Secondly the colour constancy transformation in Chapter 1 could be improved using stored data (or database as most systems described in the literature review use), to remember previous available gamut's. If a cameras spectral responses are known, they too can be used to increase the accuracy of the system by incorporating them into the temperature estimation algorithm.

With advances in fast feature lookups and colour temperature estimating techniques, most parts of the final system described in chapter 4 can be improved on in the future. Due to the modular design of the system, new algorithms can be easily inserted or updated without hindering or crashing other modules within the system.

References

[1] F. Alam, R. Mehmood, and I. Katib, "D2TFRS: An Object Recognition Method for Autonomous Vehicles Based on RGB and Spatial Values of Pixels," 2018, pp. 155–168.

[2] F. A. Spanhol, L. S. Oliveira, C. Petitjean, and L. Heutte, "Breast cancer histopathological image classification using Convolutional Neural Networks," in *2016 International Joint Conference on Neural Networks (IJCNN)*, 2016, pp. 2560–2567.

[3] R. P. Haff and N. Toyofuku, "X-ray detection of defects and contaminants in the food industry," *Sens. Instrum. Food Qual. Saf.*, vol. 2, no. 4, pp. 262–273, Dec. 2008.

[4] D. A. Mitzias and B. G. Mertzios, "A neural multiclassifier system for object recognition in robotic vision applications," *Measurement*, vol. 36, no. 3–4, pp. 315–330, Oct. 2004.

[5] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," Jun. 2015.

[6] C. Hue, J.-P. Le Cadre, and P. Perez, "A particle filter to track multiple objects," in *Proceedings 2001 IEEE Workshop on Multi-Object Tracking*, pp. 61–68.

[7] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *Int. J. Comput. Vis.*, vol. 60, no. 2, pp. 91–110, Nov. 2004.

[8] L. V. G. H. Bay, T. Tuytelaars, "SURF: Speeded Up Robust Features."

[9] N. Cornelis and K. U. Leuven, "Fast Scale Invariant Feature Detection and Matching on Programmable Graphics Hardware," 2008.

[10] S. Leutenegger, M. Chli, and R. Y. Siegwart, "BRISK: Binary Robust

invariant scalable keypoints," *Proc. IEEE Int. Conf. Comput. Vis.*, pp. 2548–2555, 2011.

[11] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "BRIEF : Binary Robust Independent Elementary Features," *Eur. Conf. Comput. Vis.*, pp. 778–792, 2010.

[12] A. Alahi, R. Ortiz, and P. Vandergheynst, "{FREAK}: Fast Retina Keypoint," *Proc. {IEEE} Int. Conf. Comput. Vis. Pattern Recognit.*, pp. 510–517, 2012.

[13] K. Mikolajczyk, K. Mikolajczyk, C. Schmid, and C. Schmid, "A performance evaluation of local descriptors," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 10, pp. 1615–1630, 2005.

[14] M. Muja and D. G. Lowe, "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration," *Int. Conf. Comput. Vis. Theory Appl. (VISAPP '09)*, pp. 1–10, 2009.

[15] M. Muja and D. G. Lowe, "Scalable Nearest Neighbour Algorithms for High Dimensional Data," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 36, no. 11, pp. 2227–2240, 2014.

[16] M. Muja and D. G. Lowe, "Fast Matching of Binary Features."

[17] S. Se, D. Lowe, and J. Little, "Global localization using distinctive visual features," *… Robot. Syst. 2002. IEEE/RSJ …*, no. October, pp. 226–231, 2002.

[18] S. Se, D. Lowe, and J. Little, "Vision-based mobile robot localization and mapping using scale-invariant features," *Proc. 2001 ICRA IEEE Int. Conf. Robot. Autom. Cat No01CH37164*, vol. 2, pp. 2051–2058, 2001.

[19] L. Juan and O. Gwun, "A Comparison of SIFT, PCA-SIFT and SURF," *Int. J. Image Process.*, vol. 3, no. 4, pp. 143–155, 2009.

[20] T. Gevers, J. Van De Weijer, and H. Stokman, "Color feature detection," *Color image Process. …*, pp. 203 – 226, 2007.

[21] K. Mikolajczyk and C. Schmid, "Performance evaluation of local descriptors.," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 10, pp.

1615–30, Oct. 2005.

[22] J. K. T. R. Gershon, A.D. Jepson, “From [R,G,B] to Surface Reflectance: Computing Color Constant Descriptors in Images,” *Perception*, vol. 755–758, 1988.

[23] S. D. Hordley, “Scene illuminant estimation: Past, present, and future,” *Color Res. Appl.*, vol. 31, no. 4, pp. 303–314, Aug. 2006.

[24] D. H. Foster, “Color constancy.,” *Vision Res.*, vol. 51, no. 7, pp. 674–700, Apr. 2011.

[25] G. D. Finlayson and S. D. Hordley, “Color constancy at a pixel.,” *J. Opt. Soc. Am. A. Opt. Image Sci. Vis.*, vol. 18, no. 2, pp. 253–64, Feb. 2001.

[26] J. van de Weijer, “Color Features and Local Structure in Images,” University of Amsterdam, 2004.

[27] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, “Speeded-Up Robust Features (SURF),” *Comput. Vis. Image Underst.*, vol. 110, no. 3, pp. 346–359, 2008.

[28] N. Zhang, “Computing Optimised Parallel Speeded-Up Robust Features (P-SURF) on Multi-Core Processors,” *Int. J. Parallel Program.*, vol. 38, no. 2, pp. 138–158, Dec. 2009.

[29] P. Del Moral, “Nonlinear filtering: Interacting particle resolution,” *Comptes Rendus l’Académie des Sci. - Ser. I - Math.*, vol. 325, no. 6, pp. 653–658, 1997.

[30] C. Hue, J. Vermaak, and M. Gangnet, “Color-Based Probabilistic Tracking,” pp. 661–675, 2002.

[31] J. A. Worthey, “Limitations of color constancy,” *J. Opt. Soc. Am. A*, vol. 2, no. 7, pp. 1014–1026, Jul. 1985.

[32] P. Wu, L. Kong, F. Zhao, and X. Li, “Particle filter tracking based on color and SIFT features,” *2008 Int. Conf. Audio, Lang. Image Process.*, pp. 932–937, 2008.

[33] M. D. Zmura, P. Colantoni, and J. Hagedorn, “Perception of Color Change,” no. November 1999, 2000.

[34] Z. Qi, R. Ting, F. Husheng, and Z. Jinlin, “Particle Filter Object Tracking Based on Harris-SIFT Feature Matching,” *Procedia Eng.*, vol. 29, pp. 924–929, 2012.

[35] H. Kandil and A. Atwan, “A Comparative Study between SIFT-Particle and SURF-Particle Video Tracking Algorithms,” *Int. J. Signal Process. Image Process. Pattern Recognit.*, vol. 5, no. 3, pp. 111–122, 2012.

[36] Y. Yan, J. Wang, C. Li, and Z. Wu, “Object Tracking Using SIFT Features in a Particle Filter,” 2011.

[37] S. D. Hordley and G. D. Finlayson, “Reevaluation of color constancy algorithm performance.,” *J. Opt. Soc. Am. A. Opt. Image Sci. Vis.*, vol. 23, no. 5, pp. 1008–20, May 2006.

[38] J. van de Weijer, T. Gevers, and A. Gijsenij, “Edge-based color constancy.,” *IEEE Trans. Image Process.*, vol. 16, no. 9, pp. 2207–14, Sep. 2007.

[39] G. D. Finlayson, S. D. Hordley, and P. M. Hubel, “Color by Correlation : A Simple , Unifying Framework for Color Constancy,” vol. 23, no. 11, pp. 1209–1221, 2001.

[40] G. D. Finlayson, M. S. Drew, and B. V. Funt, “Color constancy: generalized diagonal transforms suffice,” *J. Opt. Soc. Am. A*, vol. 11, no. 11, p. 3011, Nov. 1994.

[41] K. Barnard, V. Cardei, and B. Funt, “A comparison of computational color constancy algorithms--part I: methodology and experiments with synthesized data.,” *IEEE Trans. Image Process.*, vol. 11, no. 9, pp. 972–83, Jan. 2002.

[42] G. D. Finlayson and E. Trezzi, “Shades of Gray and Colour Constancy,” in *Twelfth Color Imaging Conference: Color Science and Engineering Systems, Technologies, and Applications*, pp. 37–41.

[43] D.A. Forsyth, “A Novel Algorithm for Colour Constancy,” *Int. J. Comput. Vis.*, vol. 5, no. 1, pp. 5–36, 1990.

[44] K. Barnard, L. Martin, A. Coath, and B. Funt, “A comparison of computational color constancy algorithms--part II: experiments with image data.,” *IEEE Trans. Image Process.*, vol. 11, no. 9, pp. 985–96, Jan. 2002.

[45] A. Gijsenij, T. Gevers, and J. Weijer, “Generalized Gamut Mapping using Image Derivative Structures for Color Constancy,” *Int. J. Comput. Vis.*, vol. 86, no. 2–3, pp. 127–139, Oct. 2008.

[46] S. Bianco, A. Bruna, F. Naccari, and R. Schettini, “Color space transformations for digital photography exploiting information about the illuminant estimation process.,” *J. Opt. Soc. Am. A. Opt. Image Sci. Vis.*, vol. 29, no. 3, pp. 374–84, Mar. 2012.

[47] U. Yang and K. Sohn, “Image-based colour temperature estimation for colour constancy,” *Electron. Lett.*, vol. 47, no. 5, p. 322, 2011.

[48] H.-C. Lee, E. J. Breneman, and C. P. Schulte, “Modeling light reflection for computer color vision,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 12, no. 4, pp. 402–409, Apr. 1990.

[49] G. M. Reinhard, E. , Khan, E.A. , Akyuz, A.O., and Johnson, “Color Imaging: Fundamentals and Applications,” *AK PETERS, Wellesley*, 2008.

[50] M. Drew and G. Finlayson, “Spectral sharpening with positivity,” *J. Opt. Soc. Am. A. Opt. Image Sci. Vis.*, vol. 17, no. 8, pp. 1361–70, Aug. 2000.

[51] H. Y. Chong, S. J. Gortler, and T. Zickler, “The von Kries Hypothesis and a Basis for Color Constancy,” *IEEE Int. Conf. Comput. Vis.*, 2007.

[52] G. D. Finlayson, M. S. Drew, and B. V Funt, “Spectral sharpening: sensor transformations for improved color constancy.,” *J. Opt. Soc. Am. A. Opt. Image Sci. Vis.*, vol. 11, no. 5, pp. 1553–63, May 1994.

[53] G. K. Kloss, “Colour Constancy using von Kries Transformations,” *Res. Lett. Inf. Math. Sci.*, vol. 13, pp. 19–33, 2009.

[54] S. Ratnasingam, S. Collins, and J. Hernández-Andrés, “Analysis of colour constancy algorithms using the knowledge of variation of correlated colour temperature of daylight with solar elevation,” *EURASIP J. Image Video*

Process., vol. 2013, no. 1, p. 14, 2013.

[55] R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programing*, no. March. 2008.

[56] J. R. Koza, F. H. Bennett, D. Andre, M. A. Keane, and F. Dunlap, "Automated synthesis of analog electrical circuits by means of genetic programming," *IEEE Trans. Evol. Comput.*, vol. 1, no. 2, pp. 109–128, 1997.

[57] J. R. Koza, F. H. Bennett, and O. Stiffelman, "Genetic Programming: Second European Workshop, EuroGP'99 G{ö}teborg, Sweden, May 26--27, 1999 Proceedings," R. Poli, P. Nordin, W. B. Langdon, and T. C. Fogarty, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 93–108.

[58] R. Poli, "Genetic Programming for Image Analysis," *Image (Rochester, N.Y.)*.

[59] C. Munteanu and A. Rosa, "Color image enhancement using evolutionary principles and the retinex theory of color constancy," *IEEE Signal Process. Soc.*, vol. 00, no. C, pp. 393–402, 2001.

[60] M. Ebner, "Evolving color constancy," *Pattern Recognit. Lett.*, vol. 27, no. 11, pp. 1220–1229, Aug. 2006.

[61] J. R. Koza, "Genetic programming as a means for programming computers by natural selection," *Stat. Comput.*, vol. 4, no. 2, pp. 87–112.

[62] R. Poli and W. B. Langdon, "Schema Theory for Genetic Programming with One-Point Crossover and Point Mutation," *Encycl. Artif. Intell.*, vol. 2, no. 3, pp. 1427–1443, 1992.

[63] Z. Zivkovic, "Improved adaptive Gaussian mixture model for background subtraction," *Proc. 17th Int. Conf. Pattern Recognition, 2004. ICPR 2004.*, vol. 2, no. 2, pp. 28–31, 2004.

[64] P. Kaewtrakulpong and R. Bowden, "An Improved Adaptive Background Mixture Model for Real- time Tracking with Shadow Detection," *Adv. Video Based Surveill. Syst.*, pp. 1–5, 2001.

[65] M. Piccardi, "Background subtraction techniques: a review," *2004 IEEE Int.*

Conf. Syst. Man Cybern. (IEEE Cat. No.04CH37583), vol. 4, pp. 3099–3104, 2004.

[66] D. S. Lee, "Effective Gaussian mixture learning for video background subtraction," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 5, pp. 827–832, 2005.

[67] A. Gijsenij and T. Gevers, "Color constancy using natural image statistics and scene semantics.," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 33, no. 4, pp. 687–98, Apr. 2011.

[68] G. D. Finlayson, S. D. Hordley, C. Lu, and M. S. Drew, "On the removal of shadows from images.," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 28, no. 1, pp. 59–68, Jan. 2006.

[69] C. Fredembach and G. Finlayson, "Simple Shadow Removal," pp. 3–6.

[70] G. D. Finlayson, M. S. Drew, and C. Lu, "Entropy Minimization for Shadow Removal," *Int. J. Comput. Vis.*, vol. 85, no. 1, pp. 35–57, May 2009.

[71] J. B. Park and W. Lafayette, "Efficient Color Representation for Image Segmentation under Non-white Illumination."

[72] A. Gijsenij, T. Gevers, and J. van de Weijer, "Computational color constancy: survey and experiments.," *IEEE Trans. Image Process.*, vol. 20, no. 9, pp. 2475–89, Sep. 2011.

[73] M. Nowostawski and R. Poli, "Parallel genetic algorithm taxonomy," *Int. Conf. Knowledge-Based Intell. Electron. Syst. Proceedings, KES*, pp. 88–92, 1999.

[74] K. Li, Q. Dai, and W. Xu, "Collaborative Color Calibration for Multi-Camera Systems," 2010.

[75] a. Ilie and G. Welch, "Ensuring color consistency across multiple cameras," *Tenth IEEE Int. Conf. Comput. Vis. Vol. 1*, pp. 1268–1275 Vol. 2, 2005.

[76] K. Kim and L. S. Davis, "Multi-camera Tracking and Segmentation of Occluded People on Ground Plane Using Search-Guided Particle Filtering,"

Eur. Conf. Comput. Vis., vol. 2006, no. 3953, pp. 98–109, 2006.

[77] F. A. Cheikh, S. K. Saha, V. Rudakova, and P. Wang, “Multi - people tracking across multiple cameras,” vol. 2, no. 1, pp. 23–33, 2012.

[78] K. Jeong and C. Jaynes, “Object matching in disjoint cameras using a color transfer approach,” *Mach. Vis. Appl.*, vol. 19, no. 5–6, pp. 443–455, May 2007.

[79] F.~Gustafsson, “Particle Filter Theory and Practice with Positioning Applications,” *IEEE Trans. Aerosp. Electron. Syst. Mag. Part II Tutorials*, vol. 25, no. 7, pp. 53–82, 2010.

[80] E. Rublee and G. Bradski, “ORB - an efficient alternative to SIFT or SURF,” 2011.